AFIT/GCE/ENG/93M-01

Design of a Hardware Discrete Event Simulation Coprocessor

THESIS

David W. Daniel
Captain, USAF

AFIT/GCE/ENG/93M-01

AFIT/GCE/ENG/93M-01

Design of a Hardware Discrete Event Simulation Coprocessor

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

David W. Daniel, B.S.

Captain, USAF

March, 1993

Acknowledgments

I would like to thank my advisor, Dr. (Lt Col) William Hobart, for providing the philosophy/direction that was much needed during this effort. I would also like to thank Major Mark Mehalic for his patience and invaluable assistance while temporarily standing in for Lt Col Hobart.

The support of some key students should also be mentioned. Capt Heinrich Rieping was very supportive during the thesis home-stretch. His support and encouragement are very much appreciated and were indispensible during this effort. I would also like to thank Capt Van Horn for his assistance in helping me better understand the SPECTRUM filters and in gathering test data for many simulations.

I would like to thank, and commend my new son, Brandon, for just recognizing me during his first ten months. I would just like to say that I am ready to be a father now.

Most of all I must thank my wife, Cathy. Her patience, strength, love, and understanding were instrumental in our success at AFIT. Without all of these traits, AFIT could of actually been much worse. I would just like to say thanks for being there for me when I wasn't there for you. Cathy, I NEED you and I LOVE you.

David W. Daniel

Table of Contents

Page

List of Figures ix

List of Tables x

Abstract xi

I. Introduction 1

1.1 Background 1

1.2 Problem 2

1.3 Summary of Current Knowledge 3

1.3.1 Discrete Event Simulation (DES) 3

1.3.2 Continuous Simulation 3

1.3.3 Combined Discrete-Continuous Simulation 4

1.4 Constraints 4

1.5 Scope 5

1.6 Standards 5

1.7 Approach/Methodology 6

1.8 Thesis Outline 6

II. Simulation Acceleration Issues 8

2.1 Introduction 8

2.2 Simulation Acceleration Techniques 8

2.2.1 Simulation Types 8

2.2.2 Simulation Constraints 9

2.2.3 Simulation Approach 10

2.3 Summary 16

Page

III. Approach/Methodology 18

3.1 Introduction 18

3.2 Structural Decomposition 18

3.2.1 Host-Node Interfacing 18

3.2.2 LP-Specific Information Storage 20

3.2.3 Next-Event List Management 21

3.2.4 Architectural Control 21

3.3 SPECTRUM Testbed 22

3.3.1 Functions 22

3.3.2 Routine design 23

3.4 Test Approach 23

3.5 Summary 23

IV. Detailed Coprocessor Design 25

4.1 Introduction 25

4.2 Component Design Approach 25

4.2.1 Design Tools 25

4.3 Host-Node Interfacing 27

4.3.1 Data Interfacing Component 27

4.3.2 Handshaking Port Device 28

4.3.3 Interrupt Handling Component 29

4.3.4 Opcode/Operand Register 30

4.3.5 Select Generation Device 30

4.4 LP-Specific Information Storage Device 31

4.4.1 Random Access Memory (RAM) Device 31

4.5 Next-Event List Management Device 32

4.5.1 Content Addressable Memory (CAM) Device 32

4.6 Architectural Control Device 37

Page

4.6.1 DES Clock Design 38

4.6.2 Mapping Random Access Memory (MRAM) Unit . . . 38

4.6.3 Microinstruction Multiplexer (MMUX) Component . . . 40

4.6.4 Microinstruction Program Counter Component 40

4.6.5 Incrementer Component 41

4.6.6 Control Store Design 41

4.6.7 Microinstruction Register 45

4.6.8 DES Opcode Decoder 45

4.6.9 R1/R2 Mux Components 46

4.6.10 R1 and R2 Decoder Components 47

4.6.11 "AND" Latch Component 47

4.6.12 General/Special-Purpose Register Bank 47

4.6.13 PATH "A" Latch Unit 49

4.6.14 PATH "B" Latch Unit 49

4.6.15 Memory Buffer Register (MBR) Component 49

4.6.16 Memory Address Register Component 51

4.6.17 Path "A" Multiplexer Component 51

4.6.18 Arithmetic Logic Unit 51

4.6.19 Zero Logic Latch 52

4.6.20 Shifter Component 52

4.6.21 Micro-Sequence Logic Component 52

4.7 Summary 53

V. Detailed Microcode Design 55

5.1 Introduction 55

5.2 DES Microcode 55

5.2.1 Startup Simulation Routine 55

5.2.2 Fetch/Decode Routine 56

Page

5.2.3 Initialize Simulation 57

5.2.4 Post Message 57

5.2.5 Get Event 57

5.2.6 Post Event 57

5.2.7 Opcode Format 57

5.2.8 Operand Format 58

5.3 Microcode Routine Execution Examples 58

5.4 Summary 63

VI. DES Coprocessor Design Test 64

6.1 Introduction 64

6.2 Design Test Methodology 64

6.3 DES Test Bench Design 65

6.4 DES Test Data 67

6.5 DES Coprocessor Design Testing 67

6.5.1 Control Store and MRAM Load 67

6.5.2 Interrupt Routine Testing 68

6.5.3 Error Routine Testing 69

6.5.4 Event Execution Testing 69

6.6 Summary 79

VII. Results and Recommendations 81

7.1 Introduction 81

7.2 Calculation Process 81

7.2.1 Hypercube Filter Averages 81

7.2.2 DES Filter Averages 83

7.2.3 System Overhead Calculation 83

7.2.4 Overall Speedup 85

Page

7.3 Recommendations 86
7.3.1 CAM Modifications 86
7.3.2 Microcode 86
7.3.3 Behavioral Components 87
7.3.4 Timing Analysis 87
7.3.5 Paradigm Support 87
7.3.6 Hardware Implementation 87
7.4 Summary 88

Appendix A. DES SPECTRUM Algorithms 89
A.1 Read-Only Control Store Procedure 89
A.2 Fetch/Decode Procedure 89
A.3 Initialize Simulation Procedures 90
A.4 Post Message Procedures 91
A.5 Get Event Procedures 92
A.6 Post Event Procedures 93

Appendix B. DES Microcode Routines 94
B.1 Read-Only Microcode 94
B.2 Fetch/Decode Microcode 97
B.3 Initialize Simulation Microcode 99
B.4 Post Message Microcode 110
B.5 Get Event Microcode 117
B.6 Post Event Microcode 126

Appendix C. DES Microcode Instruction Set 130

Appendix D. DES VHDL Behavioral and Structural Code 134

References 135

Page

Vita 137

List of Figures

Figure		Page
1.	Inter-node Communication Path	16
2.	Desired Inter-node Communication Path	17
3.	DES Component Mapping	19
4.	Status Word Configuration	28
5.	Event List Management Device	33
6.	Discrete Event Simulation Coprocessor	39
7.	Control Store Block Diagram	42
8.	General/Special-Purpose Register Configuration	48
9.	Initialize Simulation for LP 5	60
10.	The first Post Message for LP 5	60
11.	The Fourth Post Message for LP 5	61
12.	The First Get Event for LP 5	62
13.	The First Get Event for LP 5	62
14.	Carwash Configuration	74
15.	Hypercube Simulation Data	82
16.	Hypercube Total Times	84

List of Tables

Table		Page
1.	RAM Partition Layout	31
2.	CAM Control Map	33
3.	CAM Word Definition	34
4.	Input to Output mapping	40
5.	GPR Register Original Contents	50
6.	ALU Operation	51
7.	SHIFTER Operation	52
8.	MSL Input to Output Mapping	54
9.	Load Vector Format	56
10.	Opcode Formats	58
11.	Initialize Simulation Operands	59
12.	Test Bench Algorithm	66
13.	Speedup Procedures	82
14.	Cube Filter Times	83
15.	DES Microcode Routine Test Data Processing Times	83
16.	System Overhead	85
17.	Coprocessor Speedup Ratios	85
18.	Overall Speedup using Spin Loops	86

Abstract

A hardware discrete event simulation (DES) coprocessor was designed to eliminate synchronization overhead as a possible bottleneck. The target architecture is an eight node Intel iPSC/2 Hypercube, but this design has application to future CPU designs that wish to incorporate on-chip architectural features to better support parallel processor synchronization. A structural description of a general-purpose DES hardware coprocessor is given with approximately 90 percent of the components written at the gate level. The remaining components use low-level behavioral descriptions. While the DES coprocessor microcode implements the Chandy-Misra protocol, general-purpose support for a wide-range of protocols was a primary hardware design objective.

Design of a Hardware Discrete Event Simulation Coprocessor

I. Introduction

1.1 Background

Computer simulations are used in a broad range of diverse applications such as engineering, medicine, social sciences, and the military. This thesis effort is primarily concerned with its usage in the military environment. Simulations were traditionally designed for and executed on sequential processors. However, significant increases in the size and complexity of simulations over the past 20 years have resulted in simulation models "whose computational requirements cannot be reasonably satisfied with even the fastest sequential processors [17:8]."

The Air Force has a large investment in electronic hardware. As the size and complexity of these hardware components grow, so do the development costs. The Department of Defense (DoD) started the Very High Speed Integrated Circuit (VHSIC) program to encourage the development and use of high-density integrated circuits in military systems. VHSIC technology is heavily dependent on the simulation of these large, complex circuits to verify the circuit design prior to chip fabrication. Validation of circuit functionality and fault tolerance testing is essential to chip verification. This complex testing, performed through simulation, can consume months of computer time and has become a bottleneck in the logic design process [6:449].

In 1983, the VHSIC Hardware Description Language (VHDL) program was started to support standard tools required to design, test, and document large-scale circuits more efficiently and effectively. In 1987, many improvements to the VHDL language led to the IEEE Standard VHDL Language Reference Manual. VHDL has become the industry standard for simulation of large-scale circuits and also performs the important task of documentation of the circuits. Due to wide-scale acceptance of VHDL, the Department of

Defense Advanced Research Agency (DARPA) sponsored the QUEST project. The main objective of the QUEST project is a thousand-fold speedup for VHDL simulations.

In addition to the VHDL simulations required for VHSIC chip verification, a thorough timing analysis should be performed and fed back into the VHDL simulation to provide increased accuracy. Speedup of a VHDL simulation is not complete without realistic timing information to prove circuit performance. A transistor-level circuit simulator should be used to perform the timing analysis for accurate simulation models. If the circuit simulation meets the measured timing constraints, then the circuit is more likely to perform as expected.

1.2 Problem

The limitations of traditional sequential processors have led to increased interest in the area of parallel computer architectures as well as hardware simulation accelerators to increase simulation performance. The use of parallel systems has several obstacles inherent to parallel processing that must be minimized to approach maximum speedup. Among the obstacles to simulation acceleration are: the communications overhead associated with the necessary exchange of event messages between logical processes, the load imbalance of logical processes to processors, and the synchronization delay necessary to ensure event-driven simulations do not process events out of time-stamp order. The communication tasks on parallel architectures require significant simulation time and often contribute to processor idle time while the source processor waits for an acknowledgement from the destination processor. To free up the processor for event processing, a hardware coprocessor can be utilized to off-load some communication overhead. [21:6-2]

This thesis effort will use the results of the requirements analysis performed by Taylor and confront many of the remaining issues on how to implement a hardware accelerator using the conservative Chandy-Misra paradigm on a parallel multiple instruction, multiple datapath (MIMD) system [21]. The primary objective of this thesis effort is to perform a proof of concept for hardware simulation accelerators. Basically, this thesis effort shows that the synchronization overhead, associated with the passing of messages between nodes

and event management, can be off-loaded to a hardware accelerator from each of the Intel Hypercube[1] iPSC/2 80386 node processors, providing significant simulation speed-up.

1.3 Summary of Current Knowledge

Simulation models are classified by Pritsker as either discrete, continuous, or combined. The basis for this classification is how the dependent variables of the simulation model change with respect to time. Discrete simulation is further classified by the relationship between events, activities, and processes [16:63-64].

1.3.1 Discrete Event Simulation (DES) A discrete event simulation model occurs when the dependent variables change only at specified points in simulated time, referred to as event times. A DES model can be formulated by:

1. Event Orientation. Event orientation defines the changes in state that occur at event times, determines the events that can change the state of the system, and then develops the logic associated with each event type.
2. Activity Scanning Orientation. Activity scanning orientation describes entity activities in the system. The events which start or end the activity are not scheduled by the modeler, but are initiated from the conditions specified for the activity. This type of DES could be considered condition-driven.
3. Process Orientation. Process orientation describes entity flow within the system and is more directly related to standardized statements within a simulation language. The language statements are used to determine whether conditions or events have occurred, thereby signaling the need for system updating.

The objects within the discrete system are called entities. The state of the system can change only at an event time [16:63-64].

1.3.2 Continuous Simulation A continuous simulation occurs when the dependent variables can change over the entire simulation time. The dependent variables are called

[1]Hypercube is a registered trademark of the Intel Corporation.

state variables. Models of continuous systems are frequently written in terms of derivatives. Time is divided up into small time slices called steps. Continuous simulation languages for digital computers normally employ a block or statement orientation. A block orientation emulates a circuit component of an analog computer and a statement orientation models differential or difference equations [16:63-64].

1.3.3 Combined Discrete-Continuous Simulation A combined discrete-continuous simulation occurs when some dependent variables can change only at discrete times and others can change over the entire simulation time. There are two types of events that can occur in combined simulation: time-events and state-events. Time-events are those events which are scheduled to occur at specified times and state-events are those events that are not scheduled, but occur when the system reaches a particular state [16:63-64].

This thesis effort will focus on the area of discrete event simulation. The CARWASH simulation model developed by Lee will be used as a base-line for all performance measurements [12]. Along with characterizing the CARWASH simulation, Taylor developed a VHDL behavioral description of a hardware simulation accelerator, demonstrating the feasibility of improving simulation performance by off-loading the communication and synchronization overhead [21:6-2].

1.4 Constraints

This thesis effort focuses on the simulation acceleration of all discrete event simulation models; therefore, the simulation acceleration of a specific application cannot be guaranteed. A special-purpose hardware accelerator might be required for an application specific model to guarantee maximum performance gains.

All of the simulation test results gathered for this effort were compiled on the Intel iPSC/2 hypercube. The test data provides a base line to perform speedup calculations, but without realistic event processing, the simulation test data could appear biased. Therefore, speedup is quoted in terms of SPECTRUM filter speedup leading to an overall system performance gain. The amount of system performance increases can be easily changed by the length of the spin loops used to emulate the event processing time. Overall speedup

is application-dependent when using the DES coprocessor. Larger event processing time leads to decreased speed up.

All of the VHDL simulations were conducted on the AFIT VLSI network of Sun Sparc stations[2]. Each of the systems had 64 Mbytes of system memory and a variable size swap space. This constraint caused the size of the DES Content-Addressable Memory (CAM) to be down-sized to 128 words. The original target size was 1024 32-bit words. This limitation led to smaller simulation runs and less accurate results.

1.5 Scope

The goal of this thesis effort was to perform a proof of concept for off-loading synchronization overhead to a hardware simulation coprocessor. This research focuses on modeling the hardware coprocessor at the gate level; therefore, a VHDL structural description was constructed for each component of the coprocessor.

The proof of concept was documented by a VHDL structural description. The circuit design was validated through VHDL simulations, and speedup was computed when using the DES coprocessor.

1.6 Standards

The evaluation of simulation speedup is sometimes ambiguous or biased to infer the desired speedup goals are met. Logic simulation performance is rated using a different measurement criteria throughout the research. Common measurements of logic simulation performance include gate evaluations per second, instructions per second, and events per second. Simulations rated using gate evaluations per second are usually slower than those rated using events per second. Stating rates in gate evaluations per second overstates the performance since the gate evaluation rate includes the inactive gates that require no processing time [8:43]. This thesis effort compares the execution times of the discrete event simulation with and without the DES coprocessor to quantify the speedup obtained.

[2]Sparc is a registered trademark for Sun microsystems.

1.7 *Approach/Methodology*

The requirements analysis and VHDL behavioral description by Taylor provided a basis for the direction of this thesis effort [21]. The first step of this research was to perform a complete structural decomposition of the VHDL behavioral description of the hardware simulation coprocessor. The primary objective of the decomposition was to determine the feasibility of using commercial off-the-shelf (COTS) products and the possibility of using MAGIC, a chip fabrication editor, to layout some of the components.

Once the decomposition was complete, the development of a gate-level structural description using VHDL was necessary for a proof of concept. The structural description uses realistic signal propagation delays for each gate within the circuit. The propagation delays are built into the Synopsys design compiler library written by Brothers [2]. These delays were extracted from HSPICE, a timing analysis tool, runs on the respective CMOS gates. The library only provides a "NAND", "NOR", "INVERTER", and a D Flip-Flop. All of the required components can be constructed from this basic set of gates. Stringent simulation testing was conducted to ensure DES functionality would support general purpose simulations. A VHDL test bench was constructed to provide a high-level model of a Hypercube node.

Once the structural description was complete, each of the five SPECTRUM functions was written at the microcode level. The five functions implemented are *initialize simulation*, *get event*, *post event* (incoming message), *post message* (outgoing message), and *advance simulation time*. For this hardware coprocessor, the advance time function is built into the *Get Event* routine.

1.8 *Thesis Outline*

Chapter II is a synopsis of information gathered to support this research effort. Chapter III outlines the methodology used to attack this research effort and to accomplish the objectives stated in the problem statement. Chapter IV is a detailed discussion of the hardware design including the use of standard components and implementation-specific components interfaced together to obtain the functionality needed. A detailed description

of the microcode written to effectively use the DES coprocessor and implement the Chandy-Misra protocol filters is included in Chapter V. Chapter VI outlines the coprocessor test plan and the results obtained from the testing process. Chapter VII provides the thesis results and the recommendations for future actions in this area.

II. Simulation Acceleration Issues

2.1 Introduction

This chapter supplies much of the background information that was used to make decisions during the design phase. Simulation acceleration techniques are discussed in detail to provide some basic knowledge needed to understand some of the unique problems that might be encountered. Different types of simulations are discussed to provide more information required to fully support all of the functions within a given simulation type. Some of the simulation constraints are discussed to ensure an unrealistic design is not attempted. In any given simulation, software and hardware acceleration might be possible. This chapter also discusses some of the software approaches to simulation acceleration.

2.2 Simulation Acceleration Techniques

Simulation speedup is necessary to make the simulation of complex models practical. Model and implementation speedup are two methods of measuring simulation speedup. Model speedup is measured by the ratio of sequential to parallel time when the best implementation is used on both systems. This is the only speedup metric which truly reflects speedup. Implementation speedup is measured by the ratio of sequential to parallel wall-clock time when there is only one implementation of the model. [23:1-7]

To ensure speedup is stated correctly, only model speedup is considered. Stating implementation speedup could invalidate the other legitimate results of the research.

2.2.1 Simulation Types Simulation models are categorized as either discrete, continuous, or combined. State changes within the discrete simulation model can be further divided into time-driven and event-driven. The dynamic behavior of a physical system is examined by tracing various system activities as a function of time. Computer simulation models can change state only along specific time boundaries.

Time-driven simulation is considered a synchronous method. In this interval-oriented approach, time is advanced from time t to $t + \Delta t$ in uniform fixed increments of Δt. Processing of messages occurs only at the discrete time boundaries. The second method,

event-driven simulation, is asynchronous and time advances along event boundaries. Using this approach, time is "incremented from time t to the next event time t', whatever the value of t' [14:136]."

The start of the VHSIC program shifted the focus of simulation speedup in the military to logic simulation. The event-driven method is well-suited to digital logic simulation where only a small portion of the circuit, typically 10-15 percent, is active at a given time [5:67]. In the time-driven method, every time interval must be checked for candidate events. These facts reinforce the selection of the event-driven approach.

Within the area of event-driven simulation there are three major event sequencing approaches. Any of these three approaches can be used for a practical implementation.

1. Event scheduling - this approach views the system as a whole; a complete description of everything that occurs is given when an event takes place, and subsequent events are scheduled by specifying their time of occurrence.
2. Process interaction - this approach is concerned with the steps taken during the processing of an event and the interaction between the actions.
3. Activity scanning - this approach does not require an event list. An activity is defined as the state of an entity over an interval and an activity is bounded by any two successive events. This approach is more attractive than the event scheduling approach, which requires an up-to-date future events list. [14:154-155]

2.2.2 Simulation Constraints When striving for enough speedup to make a qualitative difference, some constraints limit the performance of the simulation. The basic approach to increase logic simulation speed is to write the code in assembly language. This approach usually results in less than a three-fold speedup. The next approach relies on a faster microprocessor resulting in another three-fold speedup. Combining these two approaches could result in a six- to nine-fold speedup. [3:130]

The new systems which combine the previously mentioned approaches are rated by gate evaluation speeds and event speeds. A gate evaluation represents a change in the input, while an event represents a change in the output. One event relates to approximately

2.5 evaluations. Accelerators that are rated in evaluations per second are generally much slower than those rated in events per second. When a rate is stated, the logic level of evaluation should be considered. A compiled-code simulator will appear to run faster than an event-driven simulator because the compiled-code simulator evaluates every gate at every clock pulse. A comparison can be meaningful between these two systems only when the activity level of the circuit is considered. Compiled-code simulators usually don't provide a timing analysis. [8:43-44]

Process synchronization is a necessary limitation that cannot be completely overcome. The realizability condition places the constraint of requiring processes at time t to be affected by only messages at or before time t. This requirement synchronizes the processes to ensure accurate results are obtained [13:45].

Another constraint on speedup is the problem of deadlock which occurs when using the Chandy-Misra approach to computer simulation. Deadlock occurs when all processing stops because every processor is waiting for an event that will never take place. If this problem is not resolved, the simulation cannot complete. Chandy-Misra uses null messages to eliminate this problem [4:57]. A null message is a message sent to update the time on a given input arc to possibly enable the downstream process to progress. Deadlock detection and recovery can also be used to overcome a deadlock state. Probes can be used to detect deadlock. Probes are messages sent to child nodes requesting status information [4:202]. Both approaches will work, but not without performance degradation.

2.2.3 Simulation Approach Specialized hardware and general-purpose hardware are the two prevalent approaches to hardware acceleration. Within each of these areas, proper partitioning and limiting inter-processor communications are essential to fully utilize the simulation accelerator. However, applying logical partitioning with a specialized hardware accelerator requires significant trade-offs. A general-purpose hardware approach can be designed to fully utilize a wider variety of logical partitioning methods as well as software acceleration techniques to obtain speedup over a larger range of applications.

2.2.3.1 Hardware Utilization A general-purpose hardware approach to simulation acceleration must meet many constraints to be acceptable. Two of the most important constraints to meet are simulation accuracy and flexibility. Accuracy of a simulation refers to the level of exactness obtained when comparing the physical model and the logical process. The flexibility of a simulation refers to its ability to support a variety of approaches.

D'Abreu believes that the response of the simulator, in terms of predicted signal values versus time, must correspond very closely with the response of the actual circuit [5:63]. An easy way to increase the accuracy of a model involves the use of multi-valued logic. This research effort used multi-valued logic seven (MVL7). Using various types of timing delays for all of the primitives is another way to increase a model's accuracy. This point becomes very clear during the analysis of a large circuit. If realistic timing delays are not used, then incorrect results could be obtained. [5:63-65]

A special-purpose hardware simulator can provide optimum speedup for a specific application. Therefore, the requirement for flexibility must be heavily weighted to make the general-purpose approach advantageous. The rollback chip proposed by Fujimoto is a good example of using a special-purpose hardware chip to increase the performance of a specific application [7:81].

2.2.3.2 Distributed Protocols Within the area of simulation mechanisms, there are two prevalent approaches to computer simulation. First, the Time Warp Operating System (TWOS) is considered an optimistic approach because it continues processing all incoming messages relying on rollback for process synchronization rather than waiting for all input arcs to have an event present. The second approach, the Chandy-Misra protocol is considered a conservative method since processing continues only when all input arcs have received a time-stamped message.

The Time Warp mechanism is based on the Virtual Time paradigm. Virtual Time is defined by Jefferson as a method of organizing distributed systems by imposing on them a temporal coordinate system more computationally meaningful than real time [10:404]. In this paradigm, processing continues until a message comes in with a time stamp (virtual

receive time) earlier than any message already processed and sitting in the output queue. When a message is received out-of-order, a rollback of time must occur back to the time just before that of the incoming message. To accomplish this all side effects of the messages already processed are rolled back so that the system will appears as if the messages have not yet been processed [10:405-406].

The TWOS is designed to support large-scale, irregular discrete event simulations. The TWOS runs a single simulation at a time on as many processors as are available. There are no static restrictions on the programmer. The TWOS is an event-driven mechanism that uses message passing to communicate. The messages, at a minimum, are composed of the sender, virtual send time, receiver, and virtual receive time. All messages contain a sign field which is used to identify it from its antimessage. The original message retains a positive character in the sign field and the antimessage retains a negative sign. Messages within this paradigm do not have to arrive in time-stamp order. Message processing continues until the input queue is empty. There is only one input queue for all incoming messages and one outgoing queue for all outgoing messages. Time Warp applies primarily to event-driven simulations. There are three basic mechanisms controlling the operation of this paradigm.

1. Local Control Mechanism - this mechanism controls all local processing. It executes those processes that are the oldest with respect to the current time.
2. Roll Back Mechanism - whenever a message is received with a virtual receive time in the past, the roll back mechanism starts performing the following steps: restore the last saved state before time t (new receive time), discard saved future states, and start executing messages at time t.
3. Global Control Mechanism - the global virtual time (GVT) is used to determine system progress and performs many system functions. The main concerns of the global control mechanism are: memory management, flow control, normal termination detection, error handling, I/O, snapshots, and recovery.

The GVT is responsible for removing all saved states that are earlier in time than the GVT. There must always be one saved state older than GVT to enable a process to roll back to a correct state. [10:410-419]

The actions necessary to roll back a process are achieved through the use of antimessages. For every message there is an antimessage that is exactly like the original message except for its sign. Whenever a message is sent to a receiver's input queue, an antimessage is placed in the sender's output queue. Antimessages make it possible to eliminate all side effects of a message before the simulation is adversely affected.

Whenever a message and its antimessage appear in the same queue they annihilate each other. A negative message will cause a rollback to occur at the destination if the original message has already been processed. If the original message is still present in the receiver's input queue, annihilation occurs without causing a rollback of the process. These simple rules are essential to the robust antimessage protocol. The cost of this approach is simply the cost of the rollback and antimessage overhead [10:414,416].

The Chandy-Misra algorithm maps physical processes (PP) to a distributed network of logical processes (LP) communicating via time-stamped messages. This approach requires an entry on every input arc for all communicating processes. This requirement ensures that events arriving in time-stamp order are processed in order. Any entities that interact at discrete intervals of time can be simulated by a network of processes communicating via messages. Predictability must be met by every physical system. This condition requires that for every cycle at time t there is a PP in the cycle and a real number ϵ, $\epsilon > 0$, such that the messages sent by PP along the cycle can be determined up to $t + \epsilon$ time in the future. There is a logical process corresponding to every PP. [13:45-46]

The requirement for a message on all input arcs produces a problem of simulation deadlock that must be addressed. Chandy-Misra uses the concept of null messages to avoid the deadlock problem. A process sends a message of the form (t, null) to denote the lack of a real message for the receiving process during a given time interval. A null message is also sent to all output arcs whenever a null message is received and processed by a LP. Measurements show that a large fraction of the messages sent are null messages[4:201-202].

The overhead associated with null messages can be eliminated by using a deadlock detection and recovery algorithm. However, this approach has not been proven to outperform deadlock resolution via null messages.

The deadlock detection and recovery simply consists of allowing a simulation to continuously deadlock and then recover. A special process called the controller is used to detect deadlock. The controller is then tasked to initiate a computation forcing the LPs to advance their local clocks. Although the controller is a central process, since it does not carry out any computations, it is not expected to be a bottleneck. [4:202]

2.2.3.3 Hardware Coprocessor Implementation This section outlines the functions and architectural factors that will be considered during this research effort. The areas of concern are: parallel discrete event simulation (PDES), the direct connect module (DCM), and Taylor's implementation [21].

The PDES framework is a discrete event simulation method that uses global reductions on state information to expedite the dissemination of critical information. PDESs consist of processes that communicate using time-stamped messages. A local clock is used to generate the respective timestamps of the messages in the system. Reynolds mentions the use of an auxiliary parallel reduction network (PRN) that can disseminate required global information many orders of magnitude faster than it can be disseminated in typical distributed memory multicomputers [15:167]. The following assumptions should be considered to ensure that the worst case scenario does not cause simulation failure. These assumptions are:

1. An LP can communicate with any other LP.
2. Events can be processed in zero-time.
3. Events can be preemptive.
4. Events can be spawned and consumed.

The ability to handle an event from any other LP has often been touted as a major advantage of PDES protocols that employ aggressive processing strategies [10].

The three global values used by PDES's to enhance parallel simulation are minimum next event time, smallest unreceived message, and sum. The minimum next event time, T'_η, is the next event to be executed on LP_i. The smallest unreceived message, T'_υ, is simply the timestamp of the longest outstanding message from LP_i without a receive acknowledgement. The sum, T'_ω, is the number of messages sent minus the number of messages received. [15:168-169]

The synchronization algorithm used to support the PDES has four functions: *test*, *sendmsg*, *rcvmsg*, and *rcvack*. The test function monitors the relation between its next event time, T_η, and T'_η. Whenever they are equal the LP_i can process its next event. The *sendmsg* function maintains a sequence of unacknowledged message pairs for its host LP. The *rcvmsg* function adjusts the receiving LP's T_η and sends an acknowledgement back to the sending LP. The *rcvmsg* function also decrements T_ω. The *rcvack* function removes message pairs from the sending LP's outstanding message sequence and adjusts T_υ. The key feature to this algorithm is its ability to identify the smallest next event time even when there are outstanding messages. [15:169-170]

The proposed framework provides efficient support for deadlock-free parallel simulation. This protocol, operating alone, applied to a typical PDES, would not support concurrency among LPs. This algorithm becomes most useful when the LP that can process safely needs to be determined since it promotes the use of an aggressive protocol running on top of the framework. [15:171-172]

All inter-node communication on the Intel iPSC/2 Hypercube must be sent through the DCMs. If the DES could communicate directly with a DCM, more of the communications and synchronization overhead could be eliminated, resulting in additional speedup. However, since information regarding the DCM is proprietary, work in this area was not possible. Instead, CPU interrupts are required to transfer information between nodes. Figure 1 shows the system configuration for inter-node communication with the proposed placement of the DES.

Figure 2 shows the ideal placement of the DES, connected in parallel with the DCM and the host node. This configuration would enable the DES coprocessor to receive and

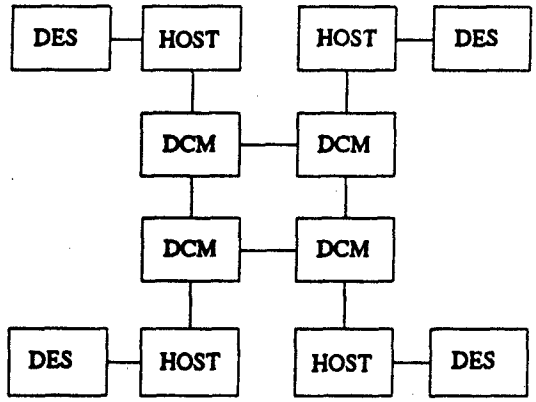

Figure 1. Inter-node Communication Path

transmit messages directly to other nodes without having to interrupt the CPU. Other computer architectures could have similar limitations on their inter-node communications. These constraints could limit speedup experienced when using a DES accelerator.

2.3 Summary

In this chapter, various approaches to simulation and the means of speeding up execution of these simulations are discussed. Time-driven and event-driven simulations are the two prominent approaches to simulation advancement. Simulation time is advanced either on discrete time boundaries or on event boundaries. The Chandy-Misra and Time Warp protocols were also discussed in order to lay out different methods of implementing simulations to ensure true operation is reflected. One of the simulation time advancement schemes must be incorporated into a given protocol to properly model a physical system. This research effort implements an event-driven simulation using the Chandy-Misra paradigm.

This research effort focuses on the design and the simulation of a general-purpose hardware accelerator which can be used to speedup simulations using a wide range of protocols. Other research efforts are focused on simulation acceleration through software

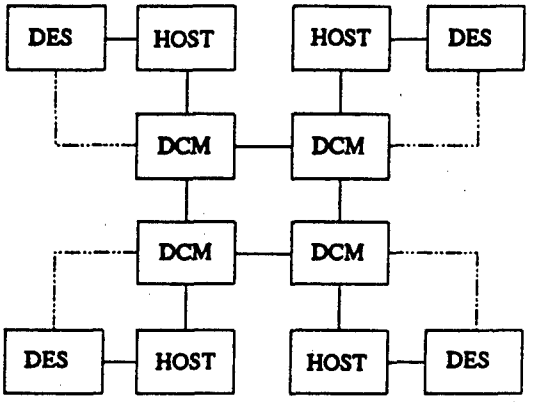

Figure 2. Desired Inter-node Communication Path

means, such as filter modifications and more effective partitioning algorithms. Many of the acceleration mechanisms can be used in combination with each other to achieve a multiplicative effect.

III. Approach/Methodology

3.1 Introduction

This chapter is an overview of the approach used to design the DES coprocessor. A structural decomposition of Taylor's code was conducted to determine and logically group the functions into components [21]. Once the decomposition was completed, the components were constructed and interfaced together to form the design. The software procedures were developed to take full advantage of the hardware design.

The software procedures are in the form of the SPECTRUM testbed filters implemented to support the Chandy-Misra protocol with null messages. Taylor's behavioral code which implemented the SPECTRUM filters was decomposed to supply the steps needed to fully support the conservative protocol [21].

Once the implementation of the design in VHDL was completed, testing procedures were developed to adequately test the design. This chapter also includes a high-level approach to the tests used in this research effort.

3.2 Structural Decomposition

The first step in the design process was to structurally decompose Taylor's behavioral VHDL code. This process resulted in a logical grouping of functions into four areas: host-node interfacing, LP-specific information storage, next-event list management, and architectural control. Figure 3 provides an overall diagram of the DES components and the system interfaces required for simulation execution. A detailed description of the components used to implement each of the four functional areas is included in Chapter IV.

3.2.1 Host-Node Interfacing Since the Intel Hypercube node uses a standard 80386 CPU for event processing, standard 80386 signal definitions were followed during the design of the DES coprocessor. There are five logical components that evolved from this requirement and they are: data interfacing, handshaking port, interrupt handling, opcode/operand determination, and select generation.

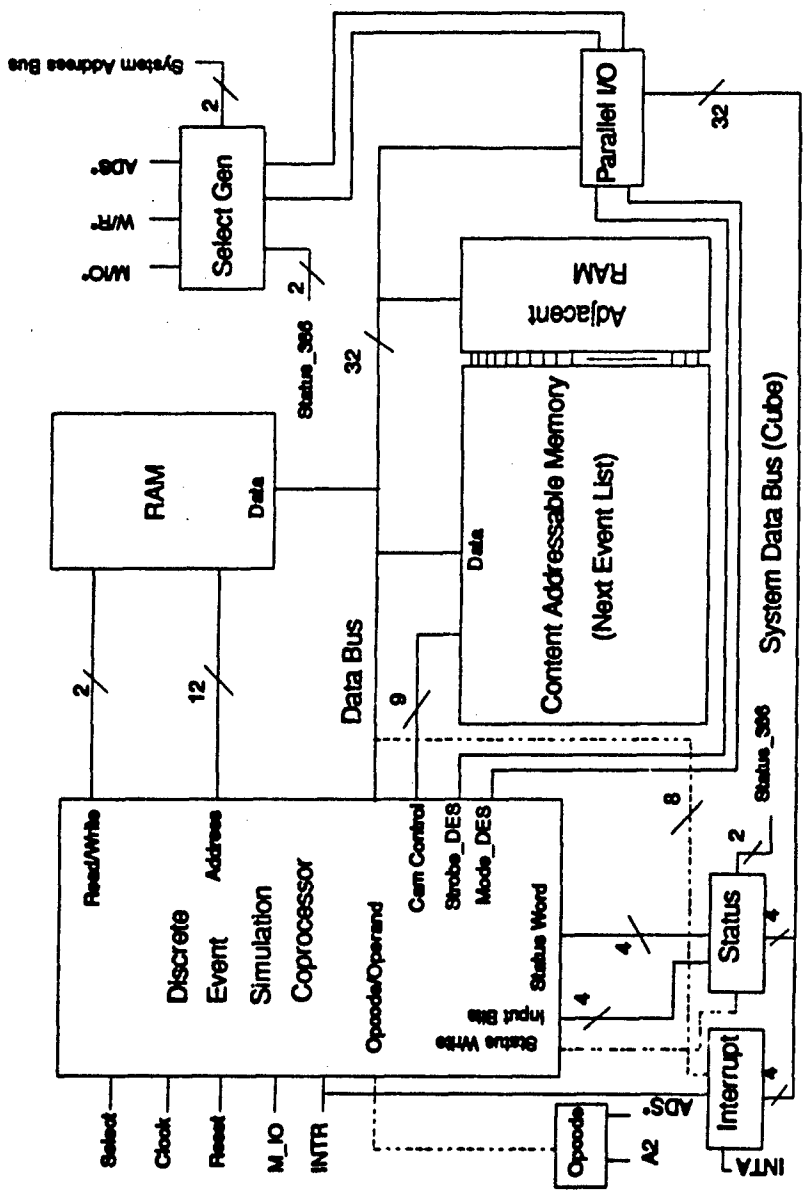

Figure 3. DES Component Mapping

3.2.1.1 Data Interfacing The DES and host system both have 32-bit buses. A parallel I/O interface was determined to be the best approach to transferring data since it allows bi-directional flow of data and can be controlled using the standard set of 80386 signals available on the Intel Hypercube.

3.2.1.2 Handshaking Port Another device used in direct support of the data interfacing device was a handshaking system. This system has to provide state information to the DES and the host system concerning the interface status. A 4-bit status register was chosen to support this requirement because the only four status parameters of concern are: ready status, error status, data ready for the DES from the host, and data ready for the host from the DES.

3.2.1.3 Interrupt Handling In order to force the host system to process DES events, an interrupt process had to be developed. Since 80386 CPU uses the lowest order eight bits of the system data bus to represent an interrupt vector, an 8-bit register to pass the vector to the system data bus was implemented.

3.2.1.4 Opcode/Operand Determination In order for the DES to distinguish between an opcode and an operand, a device had to be developed to check system address bit two which identifies the transaction type. A 1-bit register is used to hold the transaction type, opcode or operand, for DES processing purposes.

3.2.1.5 Select Generation The final requirement to properly implement the interface between the DES and the host system was the development of a component to provide chip selects for all of the interface devices. A simple combinational logic circuit is designed to use standard 80386 signals, the system data bus, and the system address bus to generate the appropriate chip selects.

3.2.2 LP-Specific Information Storage The next functional area of concern was the storage of the LP-specific information for each LP. This information is required for every filter called within the Chandy-Misra protocol. Since there are a maximum of 20 LPs per node, 20 partitions were constructed to hold the LP delay (LP_DELAY), current

simulation time (SIM_TIME), number of input and output arcs (#_I/O_ARCS), and the input and output arc encoded identification.

Static RAM, Dynamic RAM (DRAM), and DES registers were all considered for supporting this requirement. The only advantage to using DRAM is the reduction in the chip area used per cell. Because only a relatively small memory module is required, chip area was not the primary concern. Disadvantages of the DRAM are the memory refreshing circuitry and slower access times which eliminated the DRAM from consideration. Using DES registers to store the LP-specific information has many advantages. The main disadvantage, which eliminates the register approach from contention, is the chip area that would be consumed on the DES chip. An objective generated during the overall design approach was to provide maximum speedup by fabricating the DES on a single large-frame chip. Meeting this objective and placing all of these registers on-chip is not feasible; therefore, this approach was also eliminated. The final choice was the use of SRAM to maintain the LP-specific information. The SRAM is small enough and fast enough to meet the requirements of this function. Therefore, the SRAM was selected to support this requirement.

3.2.3 Next-Event List Management The next function to be considered was the retrieval of the next event for processing. This function is required every time a *Get Event* opcode is received and an event is ready. Only SRAM and a CAM were considered. The CAM was chosen because it could perform a search of its memory in O(1) time. A RAM could have been used but the search time would be at best O(log n).

3.2.4 Architectural Control In order to utilize the architecture, some control facility had to be developed. A detailed description of the subcomponents used to provide the control for the architecture and the method for supporting a wide range of protocols are discussed in Chapter IV.

3.3 *SPECTRUM Testbed*

During Taylor's requirements analysis, the primary reasons found to provide simulation acceleration was in reducing the synchronization overhead involved with event formatting, transmitting, receiving, and event-list management. Since SPECTRUM is the communications interface in use at AFIT for parallel simulation on the Intel iPSC/2 Hypercube, it was the primary target to off-load to a hardware accelerator, thereby freeing up the system for event processing. A detailed description of the implementation of the SPECTRUM filters is included in Chapter V.

SPECTRUM is the interface between the user's application program and the system-level functions. This interface enables the user to write generic simulations without concern to the architecture of the machine it will reside on. General purpose filters are used to allow the user to make system calls. Five functions are provided through the SPECTRUM filters to enable parallel simulation in a well-organized manner using the Chandy-Misra paradigm.

These functions enable the system to communicate messages (events) between LPs on the same nodes or on different nodes. Standard filters such as *Post Event* and *Post Message* are used for this type of communication between LPs. Standard filters are also useful when porting simulation programs across systems by reducing the conversion process. This provides a more general-purpose environment.

3.3.1 Functions During the decomposition of Taylor's behavioral code, the steps required for each of the five SPECTRUM filters were extracted. All of these SPECTRUM functions are supported by the DES hardware accelerator. The code is stored in the control store of the DES coprocessor and is loaded by the bootstrap ROM. This loading process is discussed in Chapter IV. The *Get Event* routine has been modified slightly to update the simulation time whenever a *Get Event* opcode is issued by the host node. The *Advance Time* function is no longer a separate function; therefore, only four functions remain to be implemented. In addition to these five functions, the *Bootstrap ROM* and *Fetch/Decode* microroutines were written to support loading of microcode and opcode

processing, respectively. The algorithms followed for each of the seven routines is located in Appendix A.

3.3.2 Routine design Some sort of phased approach to microcode execution had to be developed to standardize microinstruction processing. The microcode process for the DES is based on three phases: fetch, decode, and execute. Each of the four SPECTRUM functions has a unique opcode that points to a microroutine that controls the DES architecture. The entire microcode design is implemented with a vertical encoding to reduce the number of control bits required to perform an instruction. The microinstructions control all internal DES components through the use of an opcode decoder.

3.4 Test Approach

The DES test process was implemented in the following four areas: control store and mapping RAM loading, interrupt generation, error generation, and simulation execution. A high-level VHDL description of a 80386 CPU was implemented to enable testing in each of the areas listed. The test were also checked to ensure events occurred in a deterministic fashion. The test process and results from testing are described in more detail in Chapter VI.

3.5 Summary

First, all of the background information was gathered to provide a detailed understanding of the subject area. Next, the Chandy-Misra protocol was researched to properly implement the simulation algorithm chosen for use with the CARWASH simulation. Improper implementation of the algorithm could lead to false speedup results. Once the background information was gathered, a structural decomposition of Taylor's coprocessor was conducted to note all design decisions made during his research effort [21]. This step provided the information necessary to lay out the path to hardware accelerator completion.

The path chosen, started with the design of a behavioral description of the detailed system components. After these individual designs were thoroughly tested, generation of the VHDL structural descriptions began. After testing all of the structural components,

the micrcoroutines implementing the Chandy-Misra protocol were developed and event processing began.

IV. Detailed Coprocessor Design

4.1 Introduction

The goal of this design was to generate an efficient and effective structural description of the Discrete Event Simulation (DES) hardware accelerator with accurate timing results to prove that a hardware acclerator can provide substantial speedup.

With the design goals in place, the detailed design of the DES coprocessor is described in this chapter. This design focuses on decreasing the synchronization overhead at the node level rather than the system level. This chapter discusses the implementation of standard hardware components as well as some of the implementation-specific devices designed to meet the requirements of the Chandy-Misra protocol. The components developed as a result of the structural decomposition outlined in Chapter III are described in detail in the following subsections.

4.2 Component Design Approach

VHDL design tools were heavily used during this research. The automation process used to create a MAGIC lay out from a behavioral description is included in an OCT-TOOLS user's manual written by Kesting [11]. There are 30 steps in the automation process. The tools described in the following subsections are used in this automation process.

4.2.1 Design Tools Once a hardware accelerator architecture is designed, the implementation phase begins. There are many tools available for use at AFIT that assisted with this research effort: the Synopsys Design Compiler, EDIF2SGE program, Synopsys Simulation Graphical Environment (SGE) and Synopsys Debugger were the most effective. These tools enabled quicker design and implementation of system components than were possible by practical methods.

This research effort used only "NAND", "NOR", and "INVERTER" logic gates because these gates are faster and require fewer transistors than the "AND" and "OR" logic gates. This research was directed towards the use of a standard library of gates. This

library was supplied by Brothers as part of his dissertation [2]. The library only supported the use of the gates mentioned in addition to a "D"-type flip-flop circuit. Standardizing the gates was the first step towards automating the entire construction of the DES coprocessor.

4.2.1.1 Synopsys Design Compiler This tool was very useful for generating the VHDL structural descriptions from a simple behavioral description. There were some restrictions on the tool, such as the lack of support for case statements, variable initialization, and use of user-defined packages within a behavioral description. These limitations were easily out-weighed by the ability to produce a complete structural description in a matter of minutes. The Synopsys Design Reference Manual was the primary source for all work using this tool [18]. All of the components were designed for speed rather than area. The requirement to speed up simulation far exceeds the requirement for a smaller chip area. Standard loads and design for "worst case" conditions should ensure proper functionality of the chip at all times. All of the DES components were generated using this tool as the first step to obtaining the structural description. The design generated in the design compiler was saved in the engineering data interchange format (EDIF) to be translated by the EDIF2SGE tool.

4.2.1.2 EDIF2SGE Program This tool was used to translate the component designs into a format used by the SGE tool. This program required a script file and a configuration file to be written to identify all requirements for the translation. Both files were located in the Synopsys Simulation Graphical Environment User's Guide [19:7-18–7-20]. The script file was read into the Design Compiler for proper output file formatting and the configuration file was used in the conversion to the SGE tool format. The command was used to convert the EDIF file to the format readable by the SGE tool was *edif2sge FILENAME.edf -c configuration filename.*

4.2.1.3 Simulation Graphical Environment This tool has many capabilities that were not utilized during this research effort. This tool was only used to take the translated EDIF file and produce a VHDL structural description using a bus to represent the input and output ports when desired.

The individual component schematics were retrieved into the schematic editor and a VHDL netlist was selected for each component. This command produced all of the VHDL structural descriptions with the proper port formats. Once the designs were complete, the designs had to be checked for proper functionality.

4.2.1.4 Synopsys Debugger This tool was used to test the behavioral description prior to generation of the structural description and was also used to test the completed VHDL structural description after generation. The debugger provided the ability to trace all of the signals within a given design. This capability was effective when testing all of the internal component tests prior to connecting the system. Many design decisions were easily tested by the debugger prior to the implementation of the actual architecture.

4.2.1.5 Lager Place and Route Tool This tool automates the conversion of a file from a netlist format, which can be generated by the SGE tool, to a complete MAGIC lay out. This tool was used to complete the automation process.

4.3 Host-Node Interfacing

As outlined in Chapter III, there were five interfacing functional requirements that were confronted in this research effort. A detailed description of the components used to meet the five functional requirements is included in the following subsections.

4.3.1 Data Interfacing Component This device provides data transfers between the DES and the host system. There is one subcomponent called `pario_latch_buffer` that contains the latches and buffers required to provide 32 bits of temporary data storage. A mode signal latches the data into the "D" flip-flops and the strobe signal output enables the data onto the target bus.

An active high RESET signal is used to clear the latches whenever the DES is reset. All strobe and mode signals from the host system are generated by the select generator device. The system signals used are discussed in Section 4.3.5. A status register is required to notify the destination processor of data ready to be processed. The operation of the status register is described in Section 4.3.2.

4.3.2 Handshaking Port Device This 4-bit register provides state parameters to both of the processors. The DES is continually updated with the status. The DES and the host can change the state variables during any given clock cycle. Figure 4 shows the configuration of the four bits used to provide status information enabling handshaking between the DES and host system.

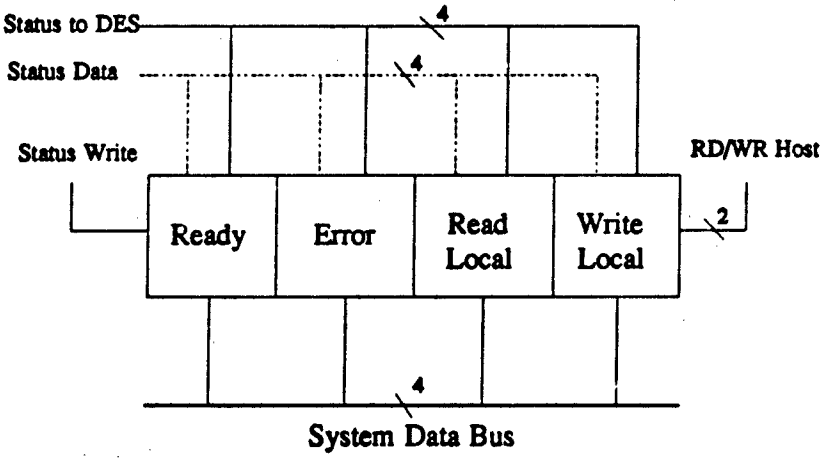

Figure 4. Status Word Configuration

Updating the status register is a three-step process. First, the requesting processor reads the status word and checks the bits of concern. Second, the processor performs the operation triggered by the value of the bit checked. Third, the processor toggles the bit of concern by performing a write to the status register with a high value on the input bit to be toggled and the status write select line activated.

Bit "0" is set high by the DES whenever it latches data to the parallel I/O device destined for the host system. An interrupt or error signal starts the transmission of data packets to the host system and the lowest-order 10 bits of the first data packet contain the count of data packets to follow the original message. Succeeding messages do not have to

be signaled with an interrupt or error bit. Bit "1" is toggled low by the host system after receiving each message from the parallel I/O device.

Bit "1" is set high by the host system whenever an opcode or operand is latched into the parallel I/O device. The host system has to check bit "3" and bit "1" before sending an opcode to the DES. If either bit is high, the host system must wait to send an opcode. If bit "1" is low, the host system can send an operand even though the DES is not in a ready state. The DES toggles bit "1" low after each message is strobed onto the local data bus and read into the coprocessor. Bit "2" is the error bit and is used in conjunction with bit "0" to send an error vector to the host system for processing. Only the DES should set the error bit high. Bit "3" is used to provide the ready status to the host system at all times. After an opcode is sent to the DES, the ready bit is set high by the DES and remains high until the opcode has been executed.

The `status_to_DES` port is designed as a direct link to the Micro-Sequence Logic Unit (MSL) component to provide state information every cycle to enable the MSL to operate efficiently. All writes by the DES are performed through the `ready_bit`, `error_bit`, `read_local`, and `write_local` bits.

4.3.3 Interrupt Handling Component This device uses the standard 386 signals interrupt request (INTR) and interrupt acknowledge (INTA). The interrupt register is divided into two processes: `INTR_LATCH` and `INTR_STROBE`. The `INTR_LATCH` process loads the 8 lower order bits of the 32-bit local data bus into the interrupt register whenever the INTR signal is high. The `INTR_STROBE` process then strobes the interrupt vector onto the system data bus for processing by the host system whenever the INTA signal is active. The output lines are placed in a high impedence state whenever the INTA signal is inactive.

An interrupt register provides a means for the DES to request processing time. An interrupt is used only to indicate to the host system that the DES has data to be transferred. There are three interrupt vectors that are used within the microcode. The *Post Event Message* interrupt notifies the host that null messages for the output arcs are ready to be processed. The *Get Event Message* interrupt is used whenever a real event message has been retrieved and is ready to be sent to the host for processing. The *Get Event Nulls*

interrupt is used whenever a null event was retrieved and the nulls have been prepared for the host to transmit.

4.3.4 Opcode/Operand Register This device is used to latch address bit "2" of the system address bus for future testing. The fetch/decode microcode routines is the only routine that should receive an opcode. All other routines are expecting operands. This address bit is latched whenever data is strobed into the parallel I/O device by the host processor. This data is constantly read by the DES for microcode branching determination.

If address bit two = "0," then the data sent to the DES is an opcode. If address bit two = "1," then the data sent to the DES is an operand. This distinction is checked several times in the microcode and will cause an error if anything except an operand is received after the initial opcode is sent. This 1-bit register was validated during the structural decomposition described in Chapter III.

4.3.5 Select Generation Device This device decodes all of the addresses and system signals and provides chip select signals to the parallel I/O, status register, and the opcode/operand register. All of the DES components that interface with the system are mapped into the I/O space in the system. Therefore, the 80386 M_IO* signal has to be low for the DES to interface with the host. A read or write is signalled by the standard 80386 WR_RD* signal. All of the signals are presumed to be invalid until the address strobe (ADS*) bit is active. The parallel I/O device is triggered whenever an I/O read or write is asserted with address bit "15" set high. The status register is triggered whenever an I/O read or write is asserted with address bit "8" set high. The opcode/operand latch is triggered whenever the parallel I/O device is triggered and the write bit is active. Simple combinational logic was used to construct the chip selects for the parallel I/O, status register, and the opcode/operand register.

Table 1. RAM Partition Layout

Bits	Description
All 32 bits used	LP Delay
All 32 bits used	Simulation Time
Two 16 bit values	#_ARCS_IN|OUT
Bits 25 downto 18 At most 10 Arcs	Input Arcs
Bits 25 downto 18 At most 10 Arcs	Output Arcs

4.4 LP-Specific Information Storage Device

The requirement for a device to maintain the simulation information specific to each LP is outlined in Section 3.2.2. The configuration and use of the SRAM is described in more detail in the following subsection.

4.4.1 Random Access Memory (RAM) Device This device provides a local memory device to maintain simulation data unique to each LP. The configuration of this component is taken from Figure 4.3 of Taylor's requirements analysis [21]. Some of the first design decisions were made concerning the RAM device. The base pointer addresses and status registers were moved to the GPRs in the DES coprocessor and the size of the CAM was increased to eliminate the need for swap space in the RAM. These enhancements to Taylor's design provided faster access to specific RAM partitions and reduced the RAM memory required by a factor of two. A typical LP partition is shown in Table 1. As mentioned in Chapter III, this device directly supports the Chandy-Misra protocol.

The read and write signals are active high. An active read or write signal with a chip select triggers the desired operation in the RAM. The highest order two bits of the **address_in** vector are used to select the RAM. These two bits provide expansion capabilities necessary to support other algorithms that require more control store and RAM for processing a simulation. Only one of the RAMs is implemented in this design and is selected whenever the two select bits are low. The data in/out port is used to

transfer data between the local data bus and the RAM device. The RAM device is divided into a memory component and a chip select component.

4.5 Next-Event List Management Device

In Section 3.2.3 the requirement for a device to manage the next-event queue for the DES is discussed. As mentioned, the CAM can perform a search in O(1) time. The configuration of the CAM and the peripheral components used to support the next-event list management function are discussed in the following subsections.

4.5.1 Content Addressable Memory (CAM) Device This device is responsible for event list management for the DES architecture. The CAM was chosen because each word in the CAM is searched in parallel. This capability provides significant speedup over other memory systems, but the *to* or *from address* was required to exploit this capability. The overall design and implementation of the CAM used in this research effort was provided by Banton as part of his dissertation research [1]. A front end driver was added to the CAM to free up the DES for other processing requirements. A cross-wiring device was also added and is described in the next section. The organization of the modified CAM, associated front-end driver, and adjacent RAM can be seen in Figure 5.

4.5.1.1 Cross-Wire I/O This device was used to cross-wire the input from the data bus to the CAM and reverse the output to the local data bus. The term "cross wiring" in this document will refer to the interconnection of a vector of the form 31 down to 0 to another component with a vector of the form 0 to 31. The DES components are all designed using bus input and output ports in the form 31 down to 0 while the CAM component designed by Banton was designed with the ports from 0 to 31 [1]. This device would not be required in a real hardware circuit and is not included in the Figure 5, but was used due to the limitations of VHDL.

4.5.1.2 Front-End Driver The front-end driver is designed to free up the DES coprocessor. The front-end driver performs five functions: *Initialize CAM*, *Find LP Minima*, *CAM Search* , *CAM Write*, and *CAM Reserve Arcs*. The corresponding control

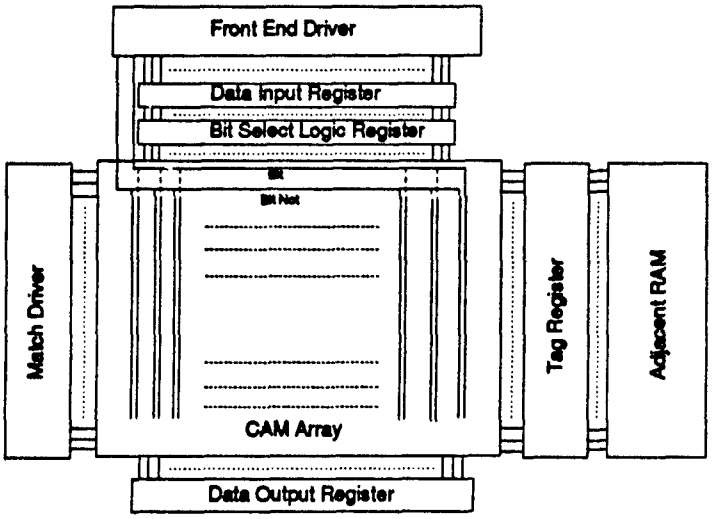

Figure 5. Event List Management Device

vectors used to trigger each of these operations are shown in Table 2. An example showing the use of each function follows in Section 5.2.

The data to be used for each of these commands has to be provided on the previous clock cycle. The data is latched into the CAM and does not change until the completion of a given CAM operation. The mask used in the CAM for bit matching is provided by the front end driver. The `CAM_COMPLETE` signal is toggled high when the operation is

Table 2. CAM Control Map

CAM Control	Operation
000001	Initialize CAM
000010	Find LP Minima
000011	CAM Search
000100	CAM Write
000101	CAM Reserve Arcs

Table 3. CAM Word Definition

CAM Bits	Field Name
0	Valid Bit
1 - 5	TO_LP
6 - 8	From Node
9 - 13	From LP
14	Reserved Arc
15 - 31	TIME_TAG

completed. The CAM also latches the corresponding address in the adjacent RAM device that is used to store the memory pointer for each event. The address is latched in to the adjacent memory input ports whenever a *CAM Write* or *Find LP Minima* is completed.

Each CAM word is broken down into fields that provide search fields for word location. Each field has specific meaning which allows the CAM's parallel search capabilities to be exploited. The 32-bit CAM words are defined in Table 3. Bit "0" is considered the most significant bit in this CAM design.

The MSL periodically checks the CAM_COMPLETE signal to continue normal operation. The CAM_MATCH flag is set appropriately depending on whether the CAM operation is a success or not. The CAM_MATCH flag is set by a logical "OR" of all of the TAG bits within the CAM. For example, if a CAM_WRITE is requested and the CAM_MATCH flag is set to a low value, then the MSL would continue operation assuming that the CAM is full.

Initialize CAM This operation is only performed during the initial load. The read-only control store memory signals the CAM to initialize through the opcode decoder. This command requires two control sequences generated by the front-end driver to complete the operation. This operation also returns a CAM_MATCH high whenever the command is completed. The TO_LP field within every CAM word is initialized to 31, and the reserved arc bit is set to zero. All words start in an unreserved mode and are changed when an input arc is reserved.

Find LP Minimum This function provides the event with the minimum TIME_TAG to the DES coprocessor. There are five steps in this process:

1. Receive TO_LP field for minima location.
2. Find the minimum for the specified LP using the **bit** and **bitnot** lines added to the CAM design. The front-end driver implements a bitwise search starting at bit "15" of the CAM word.
   (a) A search for all CAM words matching the incoming TO_LP field are performed.
   (b) The words matching the incoming TO_LP field are stored and routed to the word select lines.
   (c) A subset search using the remaining words is performed after adding Bit "15" to the search pattern. A "0" is placed in the data input register for matching.
   (d) The entire subset of words place their value on the **bit** and **bitnot** lines. A low signal on the **bit** or **bitnot** line pulls the respective line low. If no words match, the line stays at a precharged high level.
      i. If bit and bitnot = "0," then some of the words have a "0" in the searched bit and some have a "1," and the subset of words with "0" in this bit position are locked in for the next search.
      ii. If bit = "0" and bitnot = "1," then all of the words have a "0" in the bit searched and the search will move to the next bit.
      iii. If bit = "1" and bitnot = "0," then the bit searched is set to "1" and the search moves to the next bit.
      iv. If bit = "1" and bitnot = "1," then there are no matching words used in the search and the CAM_MATCH flag is set low and the CAM operation is completed. Setting the CAM_MATCH flag low will cause the calling microroutine to send an error message to the host system because this CAM function is never called unless an event is ready.
   (e) Goto step c until every bit line has a "1" or "0" and the corresponding bitnot line has the opposite value.

3. Read the CAM word to the output register. The CAM is designed to automatically select the CAM word matched that is first in the CAM array.
4. Latch the tag match bits into the corresponding address in the adjacent RAM for future memory pointer retrieval.
5. Invalidate the valid bit of the CAM word read.

The addition of the driver to recognize when all of the words in the search space have equal TIME_TAGs provides significant improvement of simulation performance as the TIME_TAGs get larger. A design deficiency was realized with the use of this feature late in the thesis cycle. If a large amount of the CAM words were pulling the **bit** or **bitnot** lines high, a single word cannot pull the corresponding line low. After this problem was realized, the CAM was modified to resolve this problem.

CAM Search This function provides a CAM search function that is used during the *Get Event* routine in the microcode. Whenever a message is retrieved, the status for a specific input arc has to be updated. This function provides a means to search for another event, on the input arc in question, to determine if the status bit should be updated. The input data received represents the FROM node and LP fields of the input arc. The CAM is searched using this data and the valid bit. The CAM_MATCH lines are automatically set. A CAM_MATCH = "0" implies there is not a matching word in the CAM. A CAM_MATCH = "1" implies there is another event on the input arc.

CAM Write This operation is responsible for all writes of validated words to the CAM. The following steps are followed to perform a CAM Write.

1. Search for a suitable CAM word with the valid bit = "0."
   (a) Search reserved arc space to determine if a reserved arc word is free for word storage.
   (b) Finally, search free CAM space.
2. Write word in the CAM.

3. Latch tag match bits into the adjacent RAM address latch for future storage of the memory pointer.
4. Set the CAM_complete bit high.
5. Set the CAM_MATCH bit high if there was a free CAM word found, else set the bit low. If the CAM_MATCH bit returned to the MSL is low, the DES must generate a CAM full error to the host system.

CAM Reserve Arcs This function is used to reserve one word in the CAM array for each input arc. This operation is only performed during simulation initialization. The following steps are used to complete this operation.

1. Search for unreserved word with the valid bit = "0."
2. Write the TO_LP field and FROM fields.
3. Toggle the reserve bit to "1."
4. Return a CAM_COMPLETE = "1" signal.

4.5.1.3 Adjacent RAM This RAM is designed to store the memory pointer for each CAM word. There is a one-to-one mapping between the adjacent RAM words and the CAM words. The address latched by the Find LP Minima and CAM Write functions is used to address the respective RAM word.

This memory latches the address whenever the ADJ_RAM_LATCH signal is high. The ADJ_RAM_control vector sent by the opcode decoder is used to trigger a read or write command. If ADJ_RAM_control = "01," then a write is performed. If ADJ_RAM_control = "10," then a read is performed. This device operates like the DES RAM device.

4.6 Architectur' Control Device

The requirement for this device can be traced back to Section 3.2.4. A detailed description of the standard and implementation specific components used to construct the DES are included in the following subsections. The d... the DES coprocessor was

taken primarily from Tannenbaum's Mic-2 architecture [20:196]. The design was enhanced to provide more speedup and ensure full support the Chandy-Misra protocol. The DES coprocessor architecture takes form in Figure 6.

4.6.1 DES Clock Design The clock for the DES design provides four-phase pulses to latch data between components internal to the DES coprocessor. The clock is critical to the DES control flow architecture. The first phase of each cycle is triggered by the external system clock of the Intel Hypercube. Since the system clock is a 25 MHz clock, the DES coprocessor is designed to run at 25 MHz to enable use of the system clock for synchronization . Each of the clock pulses is provided by the clocking and unclocking of a "D" type flip-flop. Required setup and hold times were considered when determining the proper phase lengths.

4.6.2 Mapping Random Access Memory (MRAM) Unit The MRAM operates much like the control store input mux and the control store RAM. The RAM device is loaded during the initial load routine with pointers to the start of each microroutine located in the control store RAM. There are sixty-four 10-bit words of memory in the MRAM. During normal operation, the instruction register (IR) is the source for MRAM reads. The MRAM has two components that work together to read and write the proper data into and out of memory. This device was constructed to support microroutine changes resulting in microroutine base address reassignment. Bits 31 - 26 of the instruction register (IR) are connected to the input of the MRAM to indirectly address the memory. The MRAM is loaded during initialization through ports `data_in` and `address_in`. All control, addressing, and data signals pass through the ports defined by the `MAPPING_RAM` entity declaration:

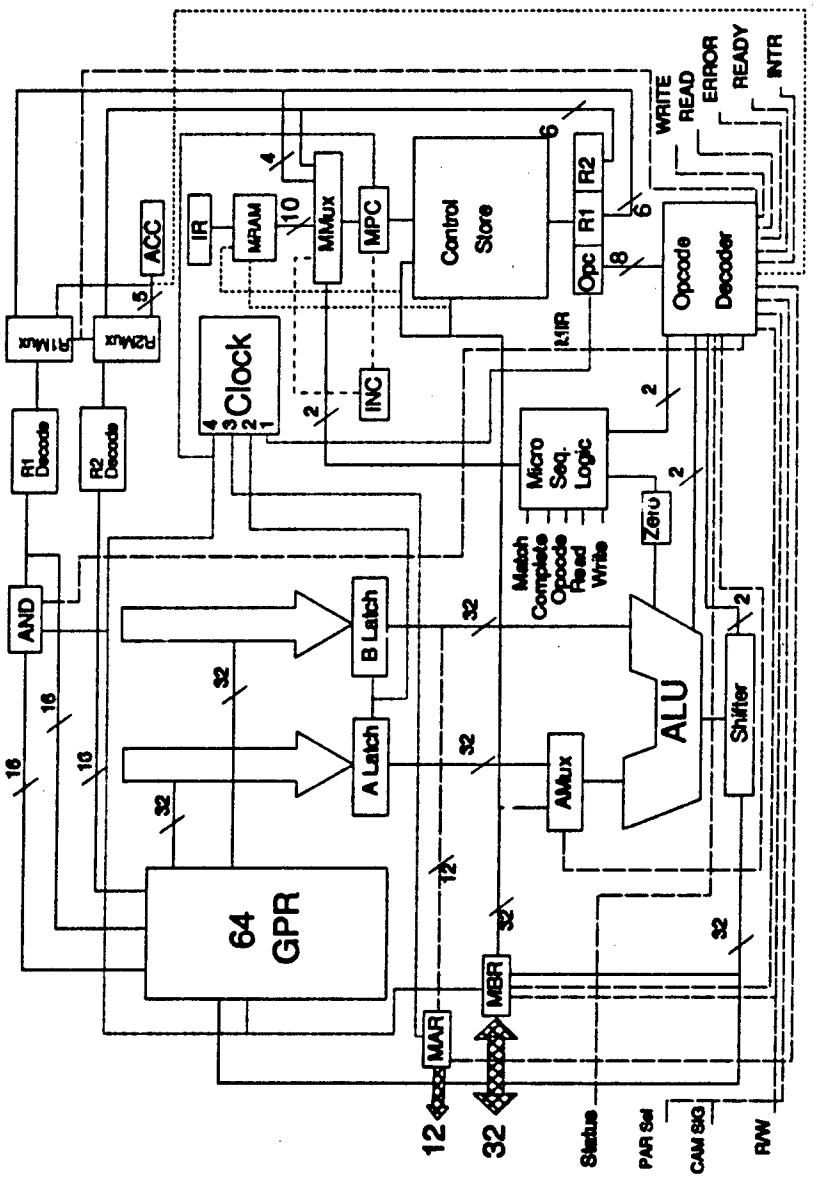

Figure 6. Discrete Event Simulation Coprocessor

Table 4. Input to Output mapping

MSL Control	Output Vector
00 or 11	Incrementer
01	Mapping ROM
10	R1 and R2

```
entity MAPPING_RAM is

  port(CHIP_ENABLE_BIT : in MVL7_vector (1 downto 0);
       IR : in MVL7_vector (5 downto 0);
       write_signal : in MVL7;
       data_in : in MVL7_vector (9 downto 0);
       address_in : in MVL7_vector (5 downto 0);
       MMUX_INPUT: out MVL7_vector (9 downto 0));

end MAPPING_ROM;
```

The first component operates like the input mux of the control store device. If `data_in_sel(1)` is high and `data_in_sel(0)` is low when the write signal is active, then the source address sent to the mapping memory is the `address_in` vector. The IR is the effective address sent to the mapping memory under all other conditions.

The second component acts like the control store RAM. If `data_in_sel(1)` is high and `data_in_sel(0)` is low when the write signal is active, this component performs a write operation. The data that is sent through the memory buffer register is stored in the mapping memory for microroutine addressing.

4.6.3 Microinstruction Multiplexer (MMUX) Component Table 4 describes the routing of the source vector to the output ports. The MMUX is triggered by the MSL device. This circuit starts the flow within the DES by sending the address of the next microinstruction to the microinstruction program counter (MPC).

4.6.4 Microinstruction Program Counter Component This device not only provides a means of addressing the control store, but also is used to signal a reload by setting the control store address to zero whenever a RESET occurs. The MPC latches the address

provided by the MMUX to the control store and the incrementer on the rising edge of the fourth clock phase. Any changes after the fourth clock phase are reflected on the next clock cycle. The RESET signal is used at start up to reset the address sent to the control store to zero. The RESET signal is active high and is generated by the select generator device whenever address bit 14 is high. This signal automatically places the DES coprocessor into the startup simulation state. The output address remains zero until the RESET signal is inactive.

4.6.5 Incrementer Component This component performs a simple binary increment of the 10-bit vector and routes the output to the MMUX as one of the three inputs for possible use as the control store microinstruction address. The incrementer is required to progress step-by-step through the microcode.

4.6.6 Control Store Design The control store provides 1024 20-bit words of control store RAM and 32 20-bit words of read-only memory (ROM) to the DES. The control was designed to provide the run time loading of a wide range of microroutines dependent on the protocol to be used in the simulation. The ROM is used to load the control store RAM with the microroutines. The control store was divided functionally into five separate components to provide modularity to the system: the input multiplexer, the chip select circuitry, the ROM, the control store RAM, and the output multiplexer. Figure 7 shows the interfaces included in this design. The interface for this component is described in the `control_store` entity declaration:

```
entity control_store is

  port(write: in MVL7;
       control_sel, address_in: in MVL7_vector (9 downto 0);
       data_in_sel: in MVL7_vector (1 downto 0);
       data_in: in MVL7_vector (19 downto 0);
       data_out: out MVL7_vector (19 downto 0));

end control_store;
```

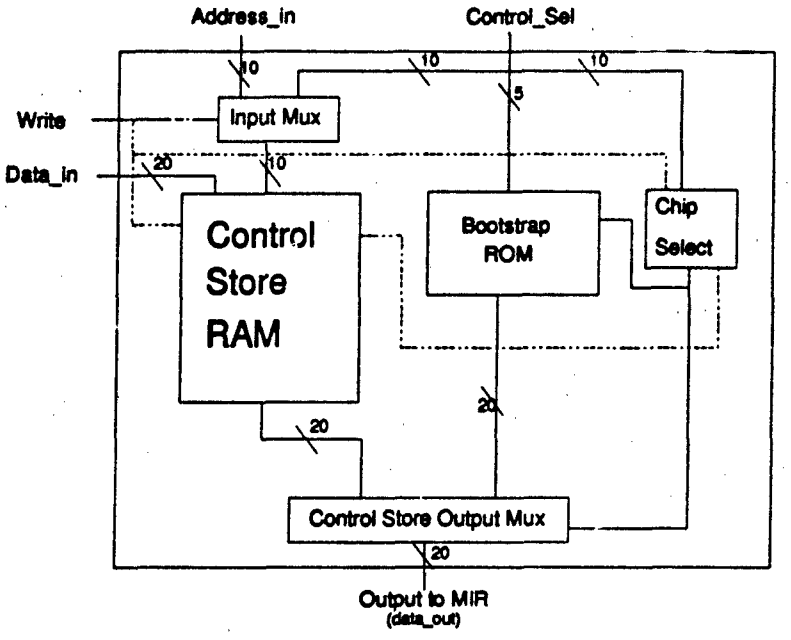

Figure 7. Control Store Block Diagram

The write signal is provided by the opcode decoder to enable control memory writes during initialization. The **address_in** lines are connected physically to the MBR lines 29 down to 20. These bits are used to specify the address for the microinstruction located in bits 19 down to 0. The microinstructions loaded into control store RAM pass through the MBR during the initial load. This process is described in Section 5.2 on the DES microcode. Table 9 shows the bit layout for an initialization vector. The MPC is responsible for selecting the next word to be read via the **control_sel** lines. The **data_in_sel** lines are used to chip select the control store RAM during a write. The **data_in** port is connected to lines 19 down to 0 of the MBR as stated above. The **data_out** port served as the input to the MIR to control the DES architecture.

4.6.6.1 Input Mux The input mux determines the source address for the control store RAM read or write. If the write signal is high and both bits of the **data_in_sel** vector are high, the source address is the **address_in** lines from the MBR; otherwise, the

control_sel lines are the source. The control_sel lines are never the source for a control store memory write.

```
entity control_mux is

  port(write : in MVL7;
       data_chip_sel: in MVL7_vector (1 downto 0);
       control_sel, addr_in: in MVL7_vector (9 downto 0);
       effective_addr : out  MVL7_vector (9 downto 0) );

end control_mux;
```

The effective_addr signal is driven by the source as determined above. This signal changes whenever either of the input addresses changed; therefore, stable addresses are required until the first clock pulse when the microinstruction is latched into the MIR. Both of these addresses were designed to be stable for at least that period.

4.6.6.2 Chip Select Circuitry The chip select circuit is responsible for providing a mem_enable and CS_mux signals to the memory components and the output mux. The memory components use the mem_enable signal and the output mux uses the CS_mux signal.

```
entity chip_sel is

port(write : in MVL7;
     control_sel: in MVL7_vector (9 downto 0);
     CS_mux, mem_enable : out MVL7 );

end chip_sel;
```

The control store RAM is enabled by a high signal on the mem_enable line. The ROM is triggered by either a high or low transition of the mem_enable line. If CS_mux is high, then the control store RAM is the source of the next microinstruction; otherwise, the ROM output is provided to the MIR as the next microinstruction.

4.6.6.3 Read-Only Memory The ROM is a bootstrap routine used to load the control store RAM during the initial load prior to the simulation initialization. There are 32 20-bit words stored in the memory. Only five bits are required for addressing this memory because there are only 32 words of memory.

4.6.6.4 Control Store RAM The writeable control store memory contains the routines responsible for supporting various algorithms through the implementation of the four SPECTRUM functions in microcode. The interface for this component occurs through the entity declaration identified by the `control_mem` entity:

```
entity control_mem is

  port(write, mem_enable : in MVL7;
       data_chip_sel : in MVL7_vector (1 downto 0);
       addr_sel : in MVL7_vector (9 downto 0);
       data_in : in MVL7_vector (19 downto 0);
       data_out: out MVL7_vector (19 downto 0));

end control_mem;
```

The `mem_enable` signal has to be high for the memory to perform a read or write of a memory word. When the `data_chip_sel` bits are both high, the DES was identified for a write during the initial load. A write occurs whenever these two conditions are true and the write signal is high. A read occurs whenever these two conditions are true and the write signal is low. During a read or write, the address is provided on the `addr_sel` lines. During a write, the data is provided through the `data_in` port. All output data is directed to the output mux via the `data_out` port.

4.6.6.5 Control Store Output This component is a simple 2-to-1 mux. The source for the output of the mux is selected from the read-only control store input vector and the writeable control memory vector. The interface to the chip select, memories, and the MIR are defined by the ports of the `control_store_out` entity declaration:

```
entity control_store_out is

  port(chip_sel: in MVL7;
       control_store_word, hard_wired_word: in MVL7_vector (19 downto 0);
       out_to_MIR : out MVL7_vector (19 downto 0));

end control_store_out;
```

If the chip_sel signal is high, then the output receives the read-only word; otherwise, the output receives the writeable control store memory word. This process is triggered by a change in any of the inputs, therefore, the input should be stable prior to and during the first clock pulse. The output of this component provides the input to the MIR.

4.6.7 Microinstruction Register This circuit provides a means of holding the selected microinstruction constant throughout a given clock cycle. The MIR is designed as a simple register with a data enable and a strobe pin. The input vector to this component is provided by the control store component. The output vector is broken down into the following three vectors: eight bits to the opcode decoder for control signal generation, six bits to the R1 mux which is used to select the input for PATH_A of the internal DES bus, and six bits to the R2 mux which is used to select the input for PATH_B on the rising edge of the first clock phase and remains latched until the next rising edge of the first phase.

4.6.8 DES Opcode Decoder This component provides most of the control signals necessary to enable proper interaction of the subcomponents within the DES architecture. This component also controls the major components outside of the DES coprocessor which includes the RAM, CAM, PARIO, adjacent RAM, status register, and the interrupt register. A list of all the microinstructions supported by the opcode decoder are included in Appendix C. The DES opcode decoder interfaces to the DES architecture through the OPCODE_DECODER entity declaration:

```
entity OPCODE_DECODER is

  port(opcode_from_MIR : in MVL7_vector (7 downto 0);
       MSL_control : out MVL7_vector (3 downto 0);
       ALU_control : out MVL7_vector (2 downto 0);
       NZ_control : out MVL7;
       SHIFTER_control : out MVL7_vector (2 downto 0);
       MBR_control : out MVL7_vector (1 downto 0);
       MAR_control : out MVL7;
       R1MUX_control : out MVL7_vector (1 downto 0);
       R2MUX_control : out MVL7_vector (1 downto 0);
       AND_LATCH_control : out MVL7;
       RAM_control : out MVL7_vector (1 downto 0);
       RAM_SEL_control : out MVL7_vector (1 downto 0);
       AMUX_control : out MVL7;
       CONTROL_STORE_control : out MVL7;
       CAM_control : out MVL7_vector (2 to 7);
       CAM_READ_control : out MVL7;
       ADJ_RAM_control  : out MVL7_vector (1 downto 0);
       INTR_control : out MVL7;
       READ_LOCAL_control : out MVL7;
       WRITE_LOCAL_control : out MVL7;
       ERROR_control : out MVL7;
       READY_control : out MVL7;
       STATUS_control : out MVL7;
       PARIO_STROBE_control : out MVL7;
       PARIO_MODE_control : out MVL7;
       PARIO_CLEAR_control : out MVL7);

end OPCODE_DECODER;
```

The opcode decoder can be implemented with a simple gate array. The input control, opcode_from_MIR, received from the MIR signals each of the outputs to a predetermined state. The output control signals are discussed in detail within each of the subcomponent descriptions.

4.6.9 R1/R2 Mux Components These components were specifically designed to provide direct access to the special-purpose registers containing the base address pointers and status registers for the LPs. These components provide the source to the R1 and R2 decoders, respectively. The accumulator (ACC) contains the LP number for a given opcode and is used to select the correct register. Only five bits of the ACC are connected

to the multiplexer. The remaining bit is sent by the opcode decoder to select the base pointer or the status register. If the control bit is low, then the base pointer is selected. If the control bit is high, the status register is selected. Figure 8 shows the complete layout of the registers within the DES.

4.6.10 R1 and R2 Decoder Components Since the DES was designed using a vertical microcode approach, these two decoder components were required to decode the register addresses for the general/special-purpose register (GPR) bank. These components provide the address of the GPR to be strobed onto the two internal data paths of the DES coprocessor. These circuits are triggered by any change in the respective decode inputs. The row and column addresses are routed to the GPR register bank. The output of the R1 decoder is also routed to the "AND" latch. The row and column vector are produced with an active low bit in the selected row or column. Eight bits of row and column address produce 64 combinations of addresses. An active low in a row and column selects the proper word. A description of the use of these input vectors is provided in Section 4.6.12.

4.6.11 "AND" Latch Component The "AND" latch was also required due the vertical nature of this von Neumann architecture. This component provides the destination address to the GPR register bank. This circuit triggers the row and column addresses to the GPR register bank on the rising edge of the fourth clock pulse. If the control bit from the DES opcode decoder is high, the GPRs perform a write to the destination register. If the control bit is low, the row and column vectors are set to all high signals. This setting effectively disables the destination write process.

4.6.12 General/Special-Purpose Register Bank This register bank provides the DES with 64 registers that are 32 bits wide. These registers are addressed by the R1 decoder, R2 decoder, and the "AND" latch. The R1 and R2 decoders provide addresses for reads onto the appropriate data paths. The "AND" latch provides a destination address for the GPR register bank. The GPR bank of registers is arranged in a 8 X 8 square. Figure 8 provides the register names and an overall view of the lay out of the registers.

(8 x 8 Register Bank)

(Rows) \ (Columns)	0	1	2	3	4	5	6	7
0	Base	Base	Base	Base	Base	Base	Base	Base
1	Base	Base	Base	Base	Base	Base	Base	Base
2	Base	Base	Base	Base				
3				TO/LP				
4	Status	Status	Status	Status	Status	Status	Status	Status
5	Status	Status	Status	Status	Status	Status	Status	Status
6	Status	Status	Status	Status	'0'	'1'	'-1'	From Mask
7	IR	ACC		TO/LP Mask	Arcs_In Mask			Count

Figure 8. General/Special-Purpose Register Configuration

The register numbers are calculated by the (*rownumber**8)+*columnnumber* formula. The row and column number are determined by the low bit in the row and column vector, respectively. The GPRs have the initial values listed in Table 5.

All of the registers are writeable. The registers are loaded during the bootstrap ROM routine along with the control store and the MRAM. Register 55 provides a mask to be used in determining the from field identity. Register 59 is also a mask register, but it is used to determine the destination LP's identity. Register 60 is the `ARCS_IN_STATUS` mask used to determine if an event is ready for processing and is also used to determine the count of operands following a given opcode. The use of all these registers will be seen in great detail in Section 5.2 on the DES microcode.

4.6.13 PATH "A" Latch Unit The only function of this unit is to latch the input from the GPRs to the multiplexer for the "A" internal data path (AMUX) on the rising edge of the second clock phase and hold the lines stable until the next clock cycle. This component requires no control from the DES opcode decoder.

4.6.14 PATH "B" Latch Unit This unit is designed to operate like a standard latch. The input data is continually read until the second clock phase. The rising edge of the second clock phase triggers the data through to the output ports and holds the data stable until the next clock cycle. This latch routes all 32 bits to the DES Arithmetic Logic Unit (ALU) and the lower order 12 bits to the Memory Address Register (MAR).

4.6.15 Memory Buffer Register (MBR) Component The MBR provides a bi-directional flow of data between the local data bus and the internal DES coprocessor. This register is used heavily during the simulation startup routine. There are separate read and write signals generated by the DES opcode decoder to select the direction of data flow. If the read signal is active, then the MBR is in the input state and reads the data off of the local data bus and routes it to the AMUX on the rising edge of the fourth clock pulse. If the write line is active, then the MBR transfers the data vector, `shifter_DATA`, to the local data bus on the rising edge of the fourth clock pulse.

Table 5. GPR Register Original Contents

Register Number	Start Value	Logical Name
1	00000000000000000000000000010111	LP1 Base Ptr
2	00000000000000000000000000101110	LP2 Base Ptr
3	00000000000000000000000001000101	LP3 Base Ptr
4	00000000000000000000000001011100	LP4 Base Ptr
5	00000000000000000000000001110011	LP5 Base Ptr
6	00000000000000000000000010001010	LP6 Base Ptr
7	00000000000000000000C0010100001	LP7 Base Ptr
8	00000000000000000000000010111000	LP8 Base Ptr
9	00000000000000000000000011001111	LP9 Base Ptr
10	00000000000000000000000G011100110	LP10 Base Ptr
11	00000000000000000000000011111101	LP11 Base Ptr
12	00000000000000000000000100010100	LP12 Base Ptr
13	00000000000000000000000100101011	LP13 Base Ptr
14	00000000000000000000000101000010	LP14 Base Ptr
15	00000000000000000000000101011001	LP15 Base Ptr
16	00000000000000000000000101110000	LP16 Base Ptr
17	00000000000000000000000110000111	LP17 Base Ptr
18	00000000000000000000000110011110	LP18 Base Ptr
19	00000000000000000000000110110101	LP19 Base Ptr
32 to 51	00000000000000000000000000000000	LP Status Regs
52	00000000000000000000000000000000	Constant "0"
53	00000000000000000000000000000001	Constant "+1"
54	11111111111111111111111111111111	Constant "-1"
55	00000000000001111111110000000000	FROM_MASK
56	00000000000000000000000000000000	Instruction Reg
57	00003000000000000000000000000000	Accumulator
59	00000011111111000000000000000000	TO_LP_MASK
60	00000000000000000000001111111111	ARCS_STATUS_MASK
63	00000000000000000000000000000000	Count Reg
Others	00000000000000000000000000000000	

Table 6. ALU Operation

ALU Control	Operation
000	Addition
001	Logical AND
010	Logical XOR
011	Logical OR
Else	Pass Through

4.6.16 Memory Address Register Component The MAR unit was designed to address the DES RAM unit when performing read and write operations. The rising edge of the third clock phase strobes the address through to the output ports, but the source input only changes when the DES opcode decoder control signal is high at any time during the clock cycle. This enhancement allows the holding of a memory address for multiple cycles, when necessary.

4.6.17 Path "A" Multiplexer Component This circuit was designed when the requirement for having multiple inputs for the same input into the ALU had to be resolved. The inputs are each 32 bits wide and the output is a 32-bit vector. The DES opcode decoder control signal determines the source vector to drive the output lines. If the control signal is low, the source vector is the MBR; otherwise, the source vector is the "A" latch.

4.6.18 Arithmetic Logic Unit The ALU is responsible for all logical and mathematical operations required in the DES. This circuitry interfaces with two 32-bit input data paths and outputs one 32-bit vector to the shifter. This circuit also sets the zero latch bit high if the result of the operation equals zero; otherwise, the zero latch bit is set to a low value.

The DES opcode decoder 3-bit control vector determines the operation to be performed by the ALU. Table 6 lists the control vectors and the related operation. All operations generate a high or low signal on the zero latch output signal. All of the operations were included only after their use was validated when writing the microcode.

Table 7. SHIFTER Operation

SHIFTER Control	Operation
000	No Shift
001	Left Shift 1 bit
010	Right Shift 1 bit
011	Left Shift 8 bits
100	Right Shift 8 bits
Else	No Shift

4.6.19 Zero Logic Latch This circuit is responsible for latching the zero latch bit from the ALU to the MSL component. The ALU signal is produced after every operation. A low signal is produced if the output of the ALU is not equal to zero and a high signal if the output equals zero. The DES opcode decoder control signal is responsible for latching the data into the MSL component. This control signal is generated by the opcode decoder every clock cycle to force the `Z_LOGIC` unit to update the zero flag state parameter in the MSL. The MSL requires this information to correctly execute the microinstructions.

4.6.20 Shifter Component The shifter unit performs four different shifts and a pass through operation. The five functions of the SHIFTER are: no shift, left shift one bit, right shift one bit, left shift eight bits, and right shift eight bits. The shifts are used throughout the microcode to either align vectors to be masked or to format output messages. The control vector breakout for shifter operation can be easily seen in Table 7.

4.6.21 Micro-Sequence Logic Component This circuitry is the primary controller used to determine the present state and the next state of the DES coprocessor. This unit is necessary to progress through the microcode correctly. This component interfaces with most of the state parameters required to determine the next logical path to follow. The opcode decoder controls the parameter checks to perform and the MSL provides the control to the MMUX depending on the parameter values found. The state parameters and control ports are defined in the following MSL entity declaration:

```
entity MSL is

   port(CAM_MATCH: in MVL7;
        CAM_COMPLETE : in MVL7;
        Z_flag: in MVL7;
        opcode_operand: in MVL7;
        READ_LOCAL_WRITE_REMOTE: in MVL7;
        WRITE_LOCAL_READ_REMOTE: in MVL7;
        MSL_control: in MVL7_vector (3 downto 0);
        MMUX_control: out MVL7_vector (1 downto 0) );

end MSL;
```

The CAM_MATCH, CAM_COMPLETE, Z_flag, opcode_operand, READ_LOCAL_WRITE_REMOTE, and WRITE_LOCAL_READ_REMOTE signals are the state parameters of the DES. The CAM provides the first two signals at the completion of each operation. The zero logic register provides the Z_flag each cycle. The details of these signals follows in the CAM, Zero Logic, and Status Word sections, respectively.

The description of the behavior of the MSL is described in Table 8 which provides an if-then type construct listing of its internal operation.

4.7 Summary

The DES coprocessor was designed with general-purpose simulation support as the primary design objective. The Chandy-Misra paradigm is implemented in microcode to provide a base for DES simulation tests. The DES is designed to potentially support many paradigms. The CARWASH simulation was used to provide test vectors to test the microcode routines and the DES interoperation.

This DES design takes the form of a standard von Neumann architecture. Every component is manipulated with control signals. The opcode decoder is the primary source of the control lines for the external and internal DES components.

The time delays built into the structural VHDL code were determined by finding the propagation delays using HSPICE. All of the propagation delays for the gates used in the DES design were obtained with HSPICE runs.

Table 8. MSL Input to Output Mapping

MSL Control	Parameter Value	MMUX Control
0000	If opcode_operand = 0	Select R1 and R2
0000	If opcode_operand = 1	Select Incrementer
0001	If opcode_operand = 0	Select R1 and R2
	and READ_LOCAL = 1	
0001	*Else*	Select Incrementer
0010	If Z_flag = 1	Select R1 and R2
0010	If Z_flag = 0	Select Incrementer
0011	If not READ_LOCAL = 1	Select R1 and R2
0011	*Else*	Select Incrementer
0100	If not WRITE_LOCAL = 1	Select R1 and R2
0100	*Else*	Select Incrementer
0101	If CAM_MATCH = 1	Select R1 and R2
0101	*Else*	Select Incrementer
0110	If CAM_COMPLETE = 1	Select R1 and R2
0110	*Else*	Select Incrementer
0111	*JUMP*	Select R1 and R2
1000	*JUMP*	Select Mapping ROM
1001	If not CAM_MATCH = 1	Select R1 and R2
1001	*Else*	Select Incrementer
1010	If not CAM_COMPLETE = 1	Select R1 and R2
1010	*Else*	Select Incrementer
1011	If not CAM_COMPLETE = 1	Select R1 and R2
	and not CAM_MATCH = 1	
1011	*Else*	Select Incrementer
Else		Select Incrementer

V. Detailed Microcode Design

5.1 Introduction

The microcode must be written to take full advantage of all the functionality built into the hardware coprocessor. The hardware developed to achieve speedup over a wide range of simulations as well as simulation protocols is of no use without the effective and efficient development of microcode to control the entire architecture. A step-by-step process was developed from the structural decomposition of the behavioral VHDL code written by Taylor [21]. The formats for the opcodes and operands is also included in this chapter to clarify the content of a given data packet.

An example follows the description of the microcode in an attempt to clarify the interaction of the microcode with the RAM, CAM, and LP status registers. The RAM, CAM, and status registers are the primary components altered by the microcode. Only snapshots of the hardware devices are included.

5.2 DES Microcode

The microcode is written to implement the five functions of the SPECTRUM testbed while providing direct support for the Chandy-Misra paradigm. As mentioned earlier, the *Advance Time* function is included in the *Get Event* routine; therefore, the five functions are implemented as four microroutines located in the control store RAM. Addresses of the starting address of each routine is stored in the mapping ROM for use by the fetch/decode routine. There are five routines located in the control store including the fetch/decode routine. Two additional routines, *Startup Simulation* and *Fetch/Decode*, were included to load the microcode into the control store and process the opcodes, respectively. All of the microcode was written using the 132 microcommands located in Appendix C. The algorithms in Appendix A were followed to write the microcode. Table 9 displays the layout of the fields and their meaning.

5.2.1 Startup Simulation Routine The control store ROM code is designed to load all of the microcode routines into the control store's RAM, initialize the CAM, initialize

Table 9. Load Vector Format

Bits	Data Vector Field
31 – 30	Chip Select
29 – 20	Address for Control Store
25 – 20	Address for Mapping RAM
19 – 00	Data for Control Store
09 – 00	Data for Mapping RAM

the general/special-purpose registers, and load the MRAM with the indirect addresses of each microcode routine. The RESET signal is designed to automatically set the address from the MPC to the control store to address "0." Once this code is called, an opcode with a value of "0" is received to start the loading of the subroutines. Loading continues until another opcode is received with a value of "0." The ROM microinstructions are contained in 32 20-bit words of control. Once this process is completed, the fetch/decode routine is called to begin normal operation.

5.2.2 Fetch/Decode Routine This microcode is designed to wait for an incoming opcode, load registers that are used by all of the subroutines, and call the desired subroutine. The opcode is stored in the accumulator because bits 25 down to 18 of the opcode contain the TO LP number. This 8-bit field is used as an input to the R1 and R2 muxes to access the base pointers and status registers. The following algorithm provides the basic flow of the **Fetch/Decode** routine.

1. Check for Opcode, loop if not.
2. Strobe data in from the Parallel Input Device.
3. Store the opcode in the IR (Register 56).
4. Load the operand count into the count register (Register 63).
5. Load the TO LP field into register 27.
6. Store the opcode in the accumulator (Register 57).
7. Jump to the address from the MRAM.

5.2.3 Initialize Simulation This routine is designed to load all of the LP specific information into the RAM, set up the status registers for each LP, send null messages to each output arc, and reserve a CAM word for each input arc. A CAM word is reserved for every input arc to ensure that at least one message can be stored in the CAM for each input arc. The opcode for this routine is 000001. This routine must be sent to the DES for each LP involved in a given simulation.

5.2.4 Post Message This routine processes all incoming messages for the host node. In general, the *Post Message* routine stores the event in the CAM, stores the memory pointer in the adjacent RAM, and updates the LP's status register. This routine is responsible for signaling a CAM_FULL whenever the CAM is full during the CAM write process. The opcode used to signal a *Post Message* is 000010.

5.2.5 Get Event The *Get Event* routine determines if there is an event ready for the LP specified in the opcode message sent by the host system, retrieves the event, sends the message to the host for processing, updates the LP's status register, and updates the simulation time for the specified LP. This function is called whenever the DES returns a CAM_FULL to the host node to free up CAM space or when an event is ready to be processed. The CAM front-end driver is used by this routine to find the event with the minimum time tag and also to search for another event matching the TO and FROM information of the event just processed. This search function is used to determine if there is another event on the same input arc. If the CAM returns a CAM_MATCH, then there is another event on the subject input arc. The opcode used to signal a *Get Event* is 000011.

5.2.6 Post Event The *Post Event* routine is only used to send null messages to all output arcs except the arc receiving the real message. The output arcs are retrieved and compared to the arc encoded in the opcode packet to determine if a null message should be formatted and sent to the host for processing. The opcode used to signal a *Post Event* is 000100.

5.2.7 Opcode Format Table 10 shows the opcodes used for each routine. All of the 32 bits are not always of use to the DES. The opcode field is stored in the IR to provide

Table 10. Opcode Formats

Instruction	Bits	Opcode Field
All Instructions	31 – 26	OPCODE Number
	25 – 23	TO_NODE
	22 – 18	TO_LP
Post Message and Event	17 – 15	FROM_NODE
Init_Sim and Get Event	17 – 15	Unused
Post Message and Event	14 – 10	FROM_LP
Init_Sim and Get Event	14 – 10	Unused
Init_Sim and Post Message	09 – 00	Operand Count
Get and Post Event	09 – 00	Unused

the address for the MRAM. The IR is used by the `fetch/decode` routine to jump to the corresponding routine in the microcode.

5.2.8 Operand Format The *Get Event* routine is the only instruction that does not require any operands to complete. The *Post Message* routine requires two operands unless the event is a null message. The `TIME_TAG` and the `MEM_PTR` are the only two operands that are expected by the *Post Message* routine. Both operands use all 32 bits of the data vector. The operand count is used to determine when the `MEM_PTR` should be set to "0" and when the `MEM_PTR` operand follows the `TIME_TAG`. The *Post Event* routine only requires one operand, the `TIME_TAG`, and all 32 bits of the data vector are used. Table 11 shows the operand format used for the initialize simulation routine.

5.3 Microcode Routine Execution Examples

The execution of the microcode routines primarily causes changes in the RAM, CAM, and LP status registers. Table 1 shows the configuration of each partition in RAM. The meaning of each field within the CAM is described in Table 3. A CAM word is valid when the "V" column as seen in Figure 9 is set to "V" and invalid when set to "N". The status register is initially set to contain a "0" for each input arc starting with a "0" in the least significant bit. The following example shows the effects on the RAM, CAM, and status

Table 11. Initialize Simulation Operands

Operand Number	Bits	Opcode Field
1	31 – 18	Unused
	17 – 15	I/O_ARC_NODE
	14 – 10	I/O_ARC_LP
	09 – 00	Unused
2	31 – 16	#_ARCS_OUT
	15 – 00	#_ARCS_IN
3	31 – 00	LP_DELAY
4	31 – 00	SIM_TIME
5	31 – 18	Unused
	17 – 00	TIME_TAG
6	31 – 00	MEM_PTR

registers after each routine is executed. The sequence begins with the *Initialize Simulation* opcode and progresses through the opcodes to the *Post Event* opcode.

Figure 9 shows the RAM, CAM, and status register contents after execution of the *Initialize Simulation* opcode sent for LP 5. The RAM has been initialized, an arc has been reserved in the CAM for each input arc, and the status register contains a "00" in the lowest order two bits.

Once all of the hardware devices have been initialized, the DES should start receiving messages to store in the CAM. Figure 10 shows the new memory contents after a message is received for LP 5. The message has been stored in the first word of the CAM. The respective CAM word has been set to the valid state. Since a message was received on the first input arc, bit "0" of the status register was updated to a "1."

Assuming three more messages have been received, what do the RAM, CAM, and status register contain? Figure 11 shows that all of the words have been stored in the CAM and their respective valid bits set. Every input arc for LP 5 has an event present in the CAM; therefore, the status bits for the input arcs are set to all 1s.

Now that there is an event ready for LP 5, Figure 12 shows the results of a *Get Event* opcode for LP 5. The RAM has been changed to reflect the new safe time of "6." The

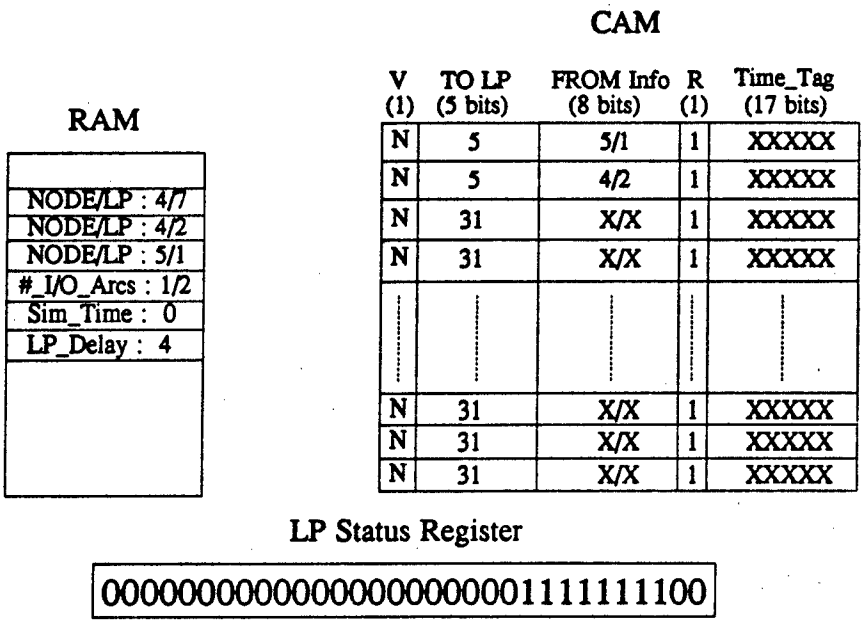

Figure 9. Initialize Simulation for LP 5

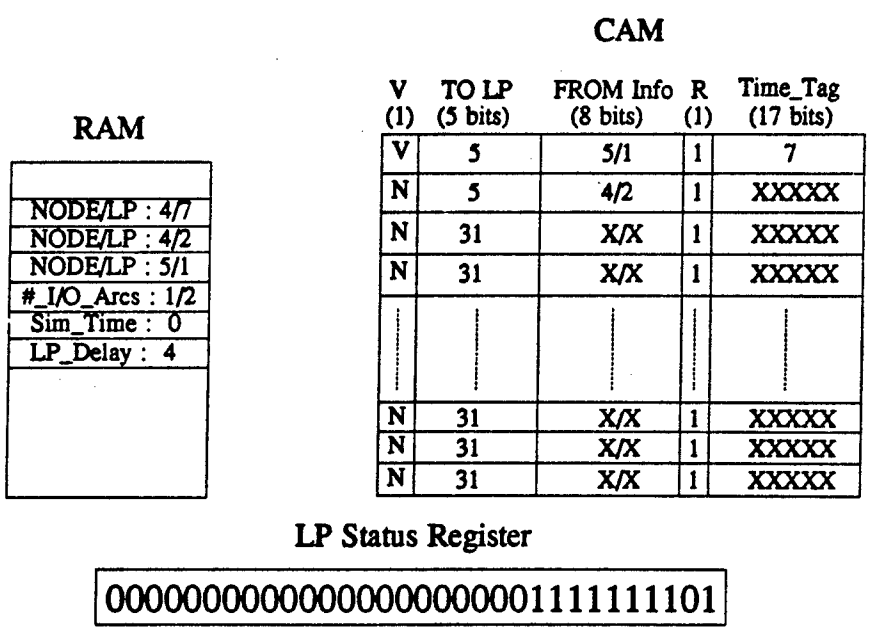

Figure 10. The first Post Message for LP 5

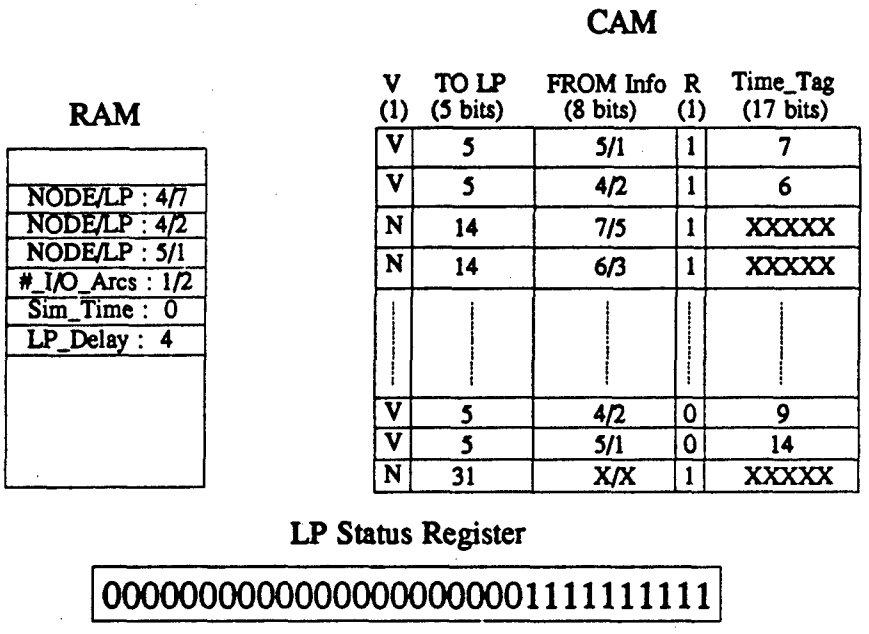

Figure 11. The Fourth Post Message for LP 5

time units are not of concern at this level. The CAM word with the smallest time tag has been retrieved and invalidated. The status register has not been changed because there is still an event on the input arc in the CAM.

Figure 13 shows the results of two *Get Events* in one step. All three of the components have been modified. The simulation time in the RAM device has been updated to "9," because the time tag of the last message retrieved contained a time tag of "9." Figure 13 shows the results of two CAM find minimum time tag commands. Since the "R" bit of the CAM is still set to a "1" for the input arcs initially reserved, the arcs remain reserved for future CAM writes.

The RAM, CAM, and status register are not updated during a *Post Event* opcode, but the information within the RAM is retrieved for processing the opcode. A null message must be formatted and sent to every output arc not receiving the real message. Figure 13 shows the location of the output arcs that must be retrieved. There is only one output arc for LP 5, but any given LP could have multiple output arcs. The `#_I/O_Arcs` field

CAM

RAM

V (1)	TO LP (5 bits)	FROM Info (8 bits)	R (1)	Time_Tag (17 bits)
V	5	5/1	1	7
N	5	4/2	1	6
N	14	7/5	1	XXXXX
N	14	6/3	1	XXXXX
⋮	⋮	⋮	⋮	⋮
V	5	4/2	0	9
V	5	5/1	0	14
N	31	X/X	1	XXXXX

RAM
NODE/LP : 4/7
NODE/LP : 4/2
NODE/LP : 5/1
#_I/O_Arcs : 1/2
Sim_Time : 6
LP_Delay : 4

LP Status Register

00000000000000000000001111111111

Figure 12. The First Get Event for LP 5

CAM

RAM

V (1)	TO LP (5 bits)	FROM Info (8 bits)	R (1)	Time_Tag (17 bits)
N	5	5/1	1	7
N	5	4/2	1	6
N	14	7/5	1	XXXXX
N	14	6/3	1	XXXXX
⋮	⋮	⋮	⋮	⋮
N	5	4/2	0	9
V	5	5/1	0	14
N	31	X/X	1	XXXXX

RAM
NODE/LP : 4/7
NODE/LP : 4/2
NODE/LP : 5/1
#_I/O_Arcs : 1/2
Sim_Time : 9
LP_Delay : 4

LP Status Register

00000000000000000000001111111101

Figure 13. The First Get Event for LP 5

would also be retrieved to locate the first output arc and to supply the count of arcs to be processed.

5.4 Summary

This chapter shows the interoperation of the hardware and microcode. The SPECTRUM filters were decomposed and routines were designed to support the Chandy-Misra protocol. Two DES routines were written to support the four filter routines described in this chapter. An example of execution of a series of opcodes and the related changes to the hardware components were detailed in this chapter to clarify the interoperation of the hardware and microcode.

The SPECTRUM filter routines designed for the DES architecture are loaded into the control store during the *Startup Simulation* process. Indirect addresses are used to jump to the correct microroutines. The microinstructions provide the control flow required to process events through the DES.

VI. DES Coprocessor Design Test

6.1 Introduction

A mixture of a behavioral and structural description of the DES coprocessor was implemented using Synopsys VHDL. All of the behavioral descriptions were written describing the behavior of low-level components, but not down to the gate level. A reference to the source code listing is located in Appendix D.

Thorough interface testing between the DES and the CPU was not possible because a working description of the Intel Hypercube iPSC/2 was not available. The interfaces were tested using 80386 signal standards as described in Volume II of Intel's Microprocessor Manual [9:5-290–5-312]. The DES was considered an I/O device with reference to the CPU, therefore, the appropriate `M_IO*` signal value was used to designate an I/O signal. The port mapping between the DES and the CPU is located in the top-level `DES_SYSTEM.vhd` file, located in Appendix D.

A very high-level VHDL test bench was designed to model the characteristics of a Hypercube node. The technique used to gather the test data for the DES design test plan is also discussed in this chapter. The final section contains the actual test cases and the results of the tests.

6.2 Design Test Methodology

The DES coprocessor design was implemented in a modular fashion. System testing was divided into hardware and microcode testing. Individual components were tested and integrated with other DES components to form logical groupings. This approach was used until all high-level units were designed and tested. The hardware integration had to be completed prior to system software integration.

The logical grouping approach to system testing resulted in eight high-level functional units. The resulting functional units are: the DES coprocessor, a parallel I/O port, a CAM, a RAM, an Interrupt register, a Status register, an Opcode/Operand latch, and the DES select generator. The DES coprocessor was further divided into 23 subcomponents

as described in Chapter IV. A whitebox test approach was followed for each of these subcomponents. The ROM routine and the *Initialize Simulation* routines were used to test the internal structure of the DES coprocessor. The parallel I/O device, the Status register, and the Opcode/Operand latch had to be integrated into the design prior to testing the control store and MRAM load process. All of the input and output ports were checked for validity during this process.

The integration testing of the parallel I/O device with the DES coprocessor was performed during the initial microcode load. The transfer of data into the DES structure was verified during the initial load of the microcode by listing the control store RAM after loading was completed. The parallel I/O ports were traced to ensure data integrity was maintained. The status bits and opcode/operand bit were checked for accuracy.

The next step was to integrate the RAM, CAM, and an interrupt register into the design to complete the DES structure. Once all of these components were implemented, the interrupt, error and event execution testing began.

Basically, the following four areas provide the testing coverage required to effectively test the implementation of the DES: control store RAM and MRAM loading, interrupt processing, error processing, and event processing. The following sections will detail the tests conducted to meet requirements.

6.3 DES Test Bench Design

This process is designed to emulate the 80386 processor at a high level. The signals generated by this design were also designed to match the signal assignments described in the Microprocessor Manual [9:5-290–5-312]. A more detailed description of the interface follows in Section 6.4. The test bench was responsible for loading the DES Control Store RAM and MRAM as described in Chapter IV. All of the signals generated in this architecture can be found in the `DES_TEST_BENCH` entity declaration:

```
entity DES_TEST_BENCH is

  port (INTR            : in DotX;
        CLK2            : out MVL7;
        SYSTEM_DATA_BUS : inout BusX (31 downto 0);
        SYSTEM_ADDRESS  : out MVL7_vector (31 downto 2);
        W_R             : out MVL7;
        M_IO            : out MVL7;
        INTA            : out MVL7;
        ADS             : out MVL7;
        RESET           : out MVL7 );

end DES_TEST_BENCH;
```

The CLK2 signal is generated by the MASTER_CLOCK process located within the **des_test_bench.vhd** file. All of the remaining I/O signals are generated or acted upon in the TRANSFER_DATA process. The signals generated by this process are triggered on the rising edge of CLK2. The DES_TEST algorithm follows the steps in Table 12.

Table 12. Test Bench Algorithm

Step	Instruction
1	Load Control Store
2	Load Mapping RAM
3	Retrieve OPCODE
4	Send OPCODE
5	Send Operands
6	Process Errors and Interrupts
7	Loop to Step 3 Until END OF FILE

The test bench was created one part at a time along with the DES coprocessor and was designed step by step in line with Table 12. As the DES coprocessor was upgraded, the test bench was upgraded to test the functionality of the coprocessor. The errors and interrupts were simply read and handled by the test bench. Checking for proper error and interrupt codes was not performed. Assertion statements were used to ensure the proper path was followed for each opcode.

6.4 *DES Test Data*

Test data from an actual Intel Hypercube run was gathered to test the DES for proper functionality using realistic data. Each of the runs produced a log file for each LP. The MAXTIME attribute in the `application.h` file can be set to the total runtime desired for a simulation. The test data was gathered using 8 LPs running on a single node of the Intel Hypercube iPSC2. These files were decoded and translated into a usable format for the test bench.

Test data for the design was gathered from a 25-second run of the "NULLWASH" simulation developed by Van Horn [22]. The Hypercube was completely unloaded when the test runs were conducted to ensure realistic filter delays were obtained. The DEBUG attribute within the `u_null_mess.c` file was turned on to print all event information for each LP. This data was used to test each individual routine of the DES coprocessor. The test data was converted into a usable format for the test bench.

Simulation average processing times per SPECTRUM filter was gathered from the "NULLWASH" simulation with a run time of 1000 seconds to ensure stable and accurate results. This data was used for comparison with the hardware implementation execution times to determine speedup. The DEBUG attribute was turned off in the `u_null_mess.c` file to turn off filter outputs. The information gathered with this configuration was the total processing time per filter, number of filter calls per filter, and the total processing time per simulation.

6.5 *DES Coprocessor Design Testing*

The DES design tests were conducted with the aid of VHDL simulations using the VHDL simulation environment and the Synopsys Debugger environment. The following four subsections contain the details of the testing results.

6.5.1 Control Store and MRAM Load The microcode written to support the Chandy-Misra protocol was used to test the initial load of the control store RAM and the MRAM. The microcode vectors were compared to the microcode and MRAM memories by listing the memories. Some of the decimal microinstructions located in the file that are loaded

into the control store RAM and MRAM were converted to hexadecimal numbers to allow a direct comparison with the control store RAM and MRAM. The values were verified to ensure the DES to host interface and the loading process was working correctly.

6.5.2 Interrupt Routine Testing The interrupt vectors checked in this test process included the following types: *Post Event Message*, which is signaled whenever a null message has been prepared for an output arc that is not receiving a real message; *Get Event Message*, which is signaled whenever an event is ready to be processed; and *Get Event Nulls*, which is used to signal that a null message was retrieved from the CAM and a null message is ready for one of the output arcs.

The test bench processed interrupts using a loop construct. An assertion statement was used within the loop construct to identify the interrupts. The code was traced during every opcode and operand to determine the interrupt routine that was selected. The interrupt register was also used to determine which of the interrupts was signaled by the DES. All of the interrupt routines are designed in the same manner to reduce coding errors. The only difference between the interrupt routines is the interrupt vector which selects the interrupt handler.

The interrupt register is the best source to review for proper interrupt signaling. The only time this register is active is when the DES has data to pass to the CPU. Testing of this component was performed by examining the interrupt vector triggered onto the system data bus during the *Initialize Simulation* routine. Both steps are recorded in the following data trace.

```
142987 NS
      M:    ACTIVE /DES_SYSTEM/DES_MAP/U6/INTR (value = '1')
     M4:    ACTIVE /DES_SYSTEM/DES_MAP/U6/INTR_VECTOR (value = X"FF")
     M2:    ACTIVE /DES_SYSTEM/DES_MAP/U6/LOCAL_DATA (value = X"FF")
143060 NS
     M1:    ACTIVE /DES_SYSTEM/DES_MAP/U6/INTA (value = '1')
     M2:    ACTIVE /DES_SYSTEM/DES_MAP/U6/LOCAL_DATA (value = X"??")
143062 NS
```

```
M3:     ACTIVE /DES_SYSTEM/DES_MAP/U6/VECTOR_TO_386 (value = X"FF")
```

The signal labeled M4 is the interrupt vector used to signal a *Post Event* interrupt. "FF" is the value of the interrupt that is processed during this interrupt. Signal M3 shows the data appearing on the interrupt outputs 2 ns after the INTA signal is activated. This interrupt validates the DES interrupt process. The entire system bus was examined to ensure the system data bus wasn't floating to a high state which would signal the same interrupt. The remaining bits of the system bus were low.

6.5.3 Error Routine Testing The error vectors examined in this process included the following types: *Should Be Operand*, which is called whenever an opcode is received by any routine except the *Fetch/Decode* routine; *No Input/Output Arcs*, which is executed whenever an LP has either no input arcs or no output arcs; *Restart Load*, which is triggered whenever the initial loading process does not complete in the correct manner; *CAM Full*, which is selected during the *Post Message* opcode if the `CAM_MATCH` flag is inactive during a CAM write signaling there is not a free CAM word for writing; and *CAM Error*, which is triggered during the *Get Event* opcode if an event is ready for the specified LP, but the CAM signals that there is not an event in the queue for the LP.

The test bench also processed the errors in a loop construct using an assertion statement to identify when an error occurred. The microinstructions were traced to ensure the appropriate error routines were executed during each test. The error routines are also designed exactly alike to reduce coding problems. Again, the vectors are the only differences between the error routines. Detailed error tests were also performed late in this research effort resulting in a limited availability of testing output, but all of the error vectors were visually checked to ensure functionality.

6.5.4 Event Execution Testing Event execution testing was performed in a modular fashion. Since the opcodes were translated from a run of a Intel Hypercube simulation, the test vectors used for the microcode tests will provide a true test of the hardware. These opcodes do not guarantee complete functionality, but provide a high level of confidence in

the operational integrity of the system. The event execution testing process includes the parallel I/O device, status register, and microroutine tests.

6.5.4.1 Parallel I/O Component Testing In this section, test results for this device are presented showing data on the input and output ports of this device. The proper port values were examined for accuracy to ensure proper bidirectional operation. The signal generator device is responsible for decoding test bench signals into the appropriate chip selects for the parallel I/O device and the status register.

The following data trace shows the CPU loading the parallel I/O device with an op-code or operand. The data is loaded into the parallel I/O device whenever the MODE_386 signal is active. This first trace segment shows a data value of "C2167029" being loaded into the parallel I/O device. Validation of this fact is seen in the next trace segment that shows the same data packet on the local data bus.

```
945 NS
  M3:    ACTIVE /DES_SYSTEM/DES_MAP/U3/MODE_386 (value = '1')
  M1:    ACTIVE /DES_SYSTEM/DES_MAP/U3/STROBE_386 (value = '0')
947 NS
  M2:    ACTIVE /DES_SYSTEM/DES_MAP/U3/MODE_DES (value = '0')
   M:    ACTIVE /DES_SYSTEM/DES_MAP/U3/STROBE_DES (value = '0')
  M5:    ACTIVE /DES_SYSTEM/DES_MAP/U3/LOCAL_DATA (value = X"????????")
960 NS
  M6:    ACTIVE /DES_SYSTEM/DES_MAP/U3/SYSTEM_DATA (value = X"C2167029")
```

The following data trace shows the data propagated through the parallel I/O device to the local data bus. Data is triggered onto the local data bus whenever the STROBE_DES signal is active. Since the data on the local and system data buses match, the parallel I/O DES receive portion of the device seems to be working.

```
1107 NS
  M2:    ACTIVE /DES_SYSTEM/DES_MAP/U3/MODE_DES (value = '0')
   M:    ACTIVE /DES_SYSTEM/DES_MAP/U3/STROBE_DES (value = '1')
  M5:    ACTIVE /DES_SYSTEM/DES_MAP/U3/LOCAL_DATA (value = X"????????")
1109 NS
  M5:    ACTIVE /DES_SYSTEM/DES_MAP/U3/LOCAL_DATA (value = X"C2167029")
```

Next, the DES transmit process was tested to ensure the parallel I/O device works properly when passing data from the DES to the host processor. The data trace below shows the data being latched into the parallel I/O device by the MODE_DES active signal. The data packet "01090401" is latched into the parallel I/O device during this test.

```
142905 NS
  M3:    ACTIVE /DES_SYSTEM/DES_MAP/U3/MODE_386 (value = '0')
  M1:    ACTIVE /DES_SYSTEM/DES_MAP/U3/STROBE_386 (value = '0')
142907 NS
  M2:    ACTIVE /DES_SYSTEM/DES_MAP/U3/MODE_DES (value = '1')
   M:    ACTIVE /DES_SYSTEM/DES_MAP/U3/STROBE_DES (value = '0')
  M5:    ACTIVE /DES_SYSTEM/DES_MAP/U3/LOCAL_DATA (value = X"01090401")
```

To ensure the data path works correctly, the data must appear on the system data bus the next time the STROBE_386 signal is active. The data trace that follows shows the output from the parallel I/O device onto the system data bus. Since the data on the local and system data buses match, the PARIO output portion of the device performs correctly.

```
143145 NS
  M3:    ACTIVE /DES_SYSTEM/DES_MAP/U3/MODE_386 (value = '0')
  M1:    ACTIVE /DES_SYSTEM/DES_MAP/U3/STROBE_386 (value = '1')
143147 NS
  M2:    ACTIVE /DES_SYSTEM/DES_MAP/U3/MODE_DES (value = '0')
   M:    ACTIVE /DES_SYSTEM/DES_MAP/U3/STROBE_DES (value = '0')
  M6:    ACTIVE /DES_SYSTEM/DES_MAP/U3/SYSTEM_DATA (value = X"01090401")
  M5:    ACTIVE /DES_SYSTEM/DES_MAP/U3/LOCAL_DATA (value = X"????????")
```

6.5.4.2 Status Register Component Testing The Status register can be updated by either the DES or the CPU. The configuration of the status register is shown in Figure 4. The operation of the Status register is described in detail in Section 4.3.2. The test results gathered show the CPU and DES updating the status word at various times. The DES and CPU can also update the status word at the same time. If both processors attempt to update the same bit in the status register, the result is a double toggle which results in no change to the status register. The value of the STATUS_to_DES value was used because the status register contents are directly connected to those signals. Whenever

the CPU is updating the status word, the WRITE_386 signal and the appropriate bits to be updated are active. Whenever the DES is updating the status word, the WRITE_DES signal and the appropriate bits to be updated are active.

The writing and reading of the status word by the CPU was tested first. The following data trace shows the contents of the status word before and after the update is executed. The STATUS_TO_386 port is connected to the lowest four bits of the system data bus. From the data trace, the CPU requested that the lowest order bit be toggled to a low value and the expected results are found. The WRITE_386 signal selects the write function for the host system. Signal M10 requests that the WRITE_LOCAL bit be toggled because the host has just completed reading a data packet from the parallel I/O device.

```
150360 NS
  M9:    ACTIVE /DES_SYSTEM/DES_MAP/U1/STATUS_TO_DES (value = X"9")
150385 NS
  M7:    ACTIVE /DES_SYSTEM/DES_MAP/U1/READ_386 (value = '0')
  M8:    ACTIVE /DES_SYSTEM/DES_MAP/U1/WRITE_386 (value = '1')
150400 NS
 M10:    ACTIVE /DES_SYSTEM/DES_MAP/U1/STATUS_TO_386 (value = X"1")
  M9:    ACTIVE /DES_SYSTEM/DES_MAP/U1/STATUS_TO_DES (value = X"8")
```

Next, the read function was tested and the following data trace shows the status transmitted to the CPU when the READ_386 signal is active. The STATUS_TO_386 vector should be the same as the STATUS_TO_DES vector. The vectors mentioned are equal, therefore, the read function of the status register is working correctly on the host side.

```
150400 NS
  M9:    ACTIVE /DES_SYSTEM/DES_MAP/U1/STATUS_TO_DES (value = X"8")
150425 NS
  M7:    ACTIVE /DES_SYSTEM/DES_MAP/U1/READ_386 (value = '1')
  M8:    ACTIVE /DES_SYSTEM/DES_MAP/U1/WRITE_386 (value = '0')
150427 NS
  M6:    ACTIVE /DES_SYSTEM/DES_MAP/U1/WRITE_DES (value = '0')
  M1:    ACTIVE /DES_SYSTEM/DES_MAP/U1/READY_BIT (value = '0')
  M2:    ACTIVE /DES_SYSTEM/DES_MAP/U1/ERROR_BIT (value = '0')
```

```
  M4:     ACTIVE /DES_SYSTEM/DES_MAP/U1/WRITE_LOCAL (value = '0')
  M3:     ACTIVE /DES_SYSTEM/DES_MAP/U1/READ_LOCAL (value = '0')
150440 NS
 M10:     ACTIVE /DES_SYSTEM/DES_MAP/U1/STATUS_TO_386 (value = X"8")
```

Only the write function on the DES side was formally tested because the status register is directly connected to the MSL in the DES coprocessor and proper functionality was seen in all of the traces. Whenever the WRITE_DES signal is active, the DES is performing a write to the status register. The contents of the status word before and after are listed below. Each of the status bits was implemented in the same fashion, therefore, the other bits do not require individual tests. Figure 4 confirms that the WRITE_LOCAL bit is the lowest-order bit and thus signal M9 shows the lowest-order bit of the status word being toggled. This example shows that the DES status word write function is operating correctly.

```
150040 NS
  M9:     ACTIVE /DES_SYSTEM/DES_MAP/U1/STATUS_TO_DES (value = X"0")
150187 NS
  M6:     ACTIVE /DES_SYSTEM/DES_MAP/U1/WRITE_DES (value = '1')
  M1:     ACTIVE /DES_SYSTEM/DES_MAP/U1/READY_BIT (value = '0')
  M2:     ACTIVE /DES_SYSTEM/DES_MAP/U1/ERROR_BIT (value = '0')
  M4:     ACTIVE /DES_SYSTEM/DES_MAP/U1/WRITE_LOCAL (value = '1')
  M3:     ACTIVE /DES_SYSTEM/DES_MAP/U1/READ_LOCAL (value = '0')
150200 NS
  M9:     ACTIVE /DES_SYSTEM/DES_MAP/U1/STATUS_TO_DES (value = X"1")
```

6.5.4.3 DES Microcode Testing The tests for the microcode routines were performed using a modular technique. The routines were tested as they appear in the following sections. All of the microcode routines are inlcuded in Appendix B. The four sections include: *Initialize Simulation*, *Post Message*, *Get Event*, and *Post Event*. Figure 14 shows the configuration of the simulation to be executed during the testing process. The input and output arcs in Figure 14 are mapped to the RAM partitions for the respective LPs.

1. Initialize Simulation

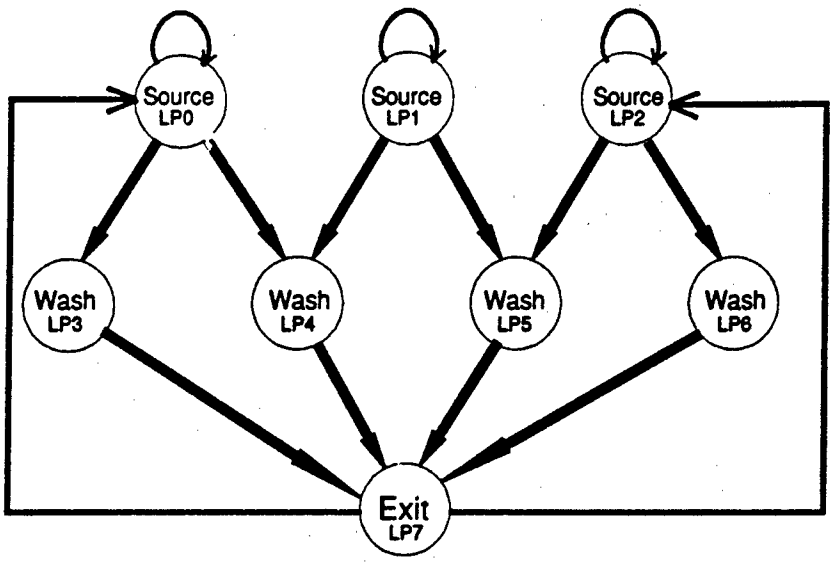

Figure 14. Carwash Configuration

As mentioned in the examples in Chapter V, there are three components to be concerned with when testing the microcode routines. The first component to be checked was the DES RAM unit. The memory listing that follows shows the LP_DELAY, SIM_TIME, I/O_ARCS, Input arcs, and output arcs as configured in Table 1. The listing shows the specific information for LP1 and LP2. The partitions are labeled below for clarification. The input and output arc information is contained in bits 25 down to 18 of the memory words. The node number is encoded in bits 25 down to 23 and the LP is encoded in bits 22 down to 18.

```
MEM_NIBBLE(
165467 NS
  M1:    ACTIVE /DES_SYSTEM/DES_MAP/U2/RAM_RW/MEM_NIBBLE (value =
 (LP1's RAM Partition:
  X"00000004", X"00000000", X"00030001", X"00000400", X"00000400",
  X"00001000", X"00001400",
  LP2's RAM Partition:
  X"00000004", X"00000000", X"00030002", X"00000800", X"00001C00",
  X"00000800", X"00001400", X"00001800",
```

The GPRs were checked next to ensure the status registers had the correct number of input arcs reflected in their respective bit vectors. There should be a 0 in each input arc bit position for every LP. The following data trace shows the GPR contents after the *initialize Simulation* routine had completed execution. In this listing of the GPRs, registers 33 and 34 contain the status registers for the specified LPs. The status registers examined are labeled below for clarification. Since register 33 has a single 0 in the lowest order bit (Hex E = "1110"), LP1 must have only one input arc. Figure 14 shows the configuration of the CARWASH simulation which confirms this fact. Register 34 corresponds to LP2 and has a 0 in the two lowest order bits. From Figure 14 register 34 is also correct. Theses examples provide a high-level of confidence in the status register routines that sets up the LP status registers.

```
165410 NS
   M:    ACTIVE /DES_SYSTEM/DES_MAP/U0/U22/GPR_REGISTERS (value =
(X"00000000", X"00000017", X"0000002E", X"00000045", X"0000005C",
 X"00000073", X"0000008A", X"000000A1", X"000000B8", X"000000CF",
 X"000000E6", X"000000FD", X"00000114", X"0000012B", X"00000142",
 X"00000159", X"00000170", X"00000187", X"0000019E", X"000001B5",
 X"081C0000", X"00080000", X"00000000", X"00000001", X"00000000",
 X"00000035", X"000003FC",
 LP1's Status Register:  X"000003FE",
 LP2's Status Register:  X"000003FC",
 X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
 X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
 X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
 X"00000000", X"00000000", X"00000000", X"00000001", X"FFFFFFFF",
 X"0003FC00", X"04080008", X"04080008", X"00000000", X"03FC0000",
 X"000003FF", X"00000000", X"00000000", X"00000001"))
```

The last component tested in this process is the CAM. The CAM array is constructed within VHDL using the generate command. This command does not allow listing of the contents of the memory, therefore, the CAM could not be listed for documentation purposes. The word select lines were observed during several reads and writes to ensure the events were placed in the correct locations. The following listing shows an input for LP0 being reserved in the first word of the CAM. The bit string has a

1 located in the highest order bit which corresponds to the reservation of the first input arc.

```
144190 NS
  M2:    ACTIVE /DES_SYSTEM/DES_MAP/TAG_ADDRESS (value = X"80000000")
```

2. Post Message

The RAM unit remains unchanged during the execution of the *Post Message* routine; therefore, the LP status registers and CAM information were examined during testing. First, the LP status registers are listed after an event is written. The information in the *Initialize Simulation* listings can serve as the state of the hardware prior to the execution of the *Post Message* opcode. The data extraction below shows the LP status register for LP1 after receiving an event on its only input arc. The status register now contains a 1 in every input arc bit position signifying the presence of an event on every input arc. The status register examined is labeled below for clarification.

```
228690 NS
   M:    ACTIVE /DES_SYSTEM/DES_MAP/U0/U22/GPR_REGISTERS (value =
(X"00000000", X"00000017", X"0000002E", X"00000045", X"0000005C",
 X"00000073", X"0000008A", X"000000A1", X"000000B8", X"000000CF",
 X"000000E6", X"000000FD", X"00000114", X"0000012B", X"00000142",
 X"00000159", X"00000170", X"00000187", X"0000019E", X"000001B5",
 X"00000001", X"00000002", X"00000400", X"0000000A", X"04040000",
 X"00000400", X"00000000", X"03FC0000", X"00000003", X"00000004",
 X"00000001", X"0000001A", X"000003FC",
 LP1's Status Register:  X"000003FF",
 X"000003FC", X"000003FF", X"000003FC", X"000003FC", X"000003FE",
 X"000003F0", X"00000000", X"00000000", X"00000000", X"00000000",
 X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
 X"00000000", X"00000000", X"00000000", X"00000000", X"00000001",
 X"FFFFFFFF", X"0003FC00", X"08040402", X"00000000", X"00000000",
 X"03FC0000", X"000003FF", X"00000000", X"00000000", X"000003FF"))
```

To ensure the proper data is being written into the CAM, the CAM_Write assertion, the local data bus, and the tag address were examined. The data trace that follows

shows the assertion and states of each signal of concern. All of these values were decomposed and validated. Each CAM word is directly mapped to a bit of the TAG_ADDRESS bit vector where the most-significant bit is mapped to the first word in the CAM. A "1" in any bit represents a match of the CAM word during a search. The TAG_ADDRESS in the data trace represents the selection of word 6 in the CAM. The sixth word of the CAM was reserved for input arc LP0 of LP3. Figure 10 shows the configuration of the CAM bits. The "C" in the LOCAL_DATA_BUS bit vector corresponds to the lowest order 4 bits of the TO LP field. If the string is converted to decimal, the TO LP = 3. The lowest order nibble corresponds to the time tag for the event.

```
206827 NS
Assertion WARNING at 206827 NS in design unit BEHAVE from process
/DES_SYSTEM/DES_MAP/U5/CAM_DRIVER/DRIVER:
    "CAM ENTERED"
  M3:  ACTIVE /DES_SYSTEM/DES_MAP/LOCAL_DATA_BUS (value = X"CC000005")
206828 NS
Assertion WARNING at 206828 NS in design unit BEHAVE from process
/DES_SYSTEM/DES_MAP/U5/CAM_DRIVER/DRIVER:
    "CAM WRITE"
206910 NS
  M2:  ACTIVE /DES_SYSTEM/DES_MAP/TAG_ADDRESS (value = X"04000000")
```

This data extraction validates CAM functionality using the *Post Message* routine. These test provides a sufficiently high-level of confidence in the interfacing of the hardware and the *Post Message* routine.

3. Get Event

This routine was by far the most complex of the four filters implemented in microcode. Basically, every component must work properly to obtain accurate results for the *Get Event* routine. A *Get Event* for LP1 was chosen to show a flow of events for testing validation; therefore, this *Get Event* opcode was designated for LP1. The first component to be checked during a *Get Event* opcode is the LP status register. The data trace from the *Post Message* routine shows the status register for LP1 extracted

and labeled for clarification. The register contains a 1 for every input arc bit which indicates that an event is ready for processing. LP1 has multiple inputs on the input arc; therefore, the status register does not change value. The data trace below shows that LP1's status register did not change value.

```
267130 NS
       M:    ACTIVE /DES_SYSTEM/DES_MAP/U0/U22/GPR_REGISTERS (value =
(X"00000000", X"00000017", X"0000002E", X"00000045", X"0000005C",
 X"00000073", X"0000008A", X"000000A1", X"000000B8", X"000000CF",
 X"000000E6", X"000000FD", X"00000114", X"0000012B", X"00000142",
 X"00000159", X"00000170", X"00000187", X"0000019E", X"000001B5",
 X"00000002", X"04040000", X"000003FF", X"00000004", X"04000000",
 X"0003FFFF", X"00000006", X"00040000", X"001C0000", X"04040000",
 X"00000001", X"00000018", X"000003FF",
 LP1's Status Register: X"000003FF",
 X"000003FD", X"000003FF", X"000003FC", X"000003FC", X"000003FE",
 X"000003FE", X"00000000", X"00000000", X"00000000", X"00000000",
 X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
 X"00000000", X"00000000", X"00000000", X"00000000", X"00000001",
 X"FFFFFFFF", X"0003FC00", X"0C040000", X"0C040000", X"00000000",
 X"03FC0000", X"000003FF", X"00000000", X"00000000", X"00000000"))
```

The CAM is the next component to be tested. The TAG_ADDRESS corresponds to the CAM word selected. LP1 has two input arcs and LP1 has one input arc. The arcs were reserved in order from LP0 to LP7. The TAG_ADDRESS value below corresponds to the third word in the CAM. Reviewing the CAM configuration, the value on the local data bus below represents an event from LP1 to LP1 with a time tag of 0. This word selected was a reserved word for LP1 and provides validation for the CAM find minimum time tag function because 0 is the smallest time tag.

```
266314 NS
 M2:  ACTIVE /DES_SYSTEM/DES_MAP/TAG_ADDRESS (value = X"20000000")
266549 NS
 M3:  ACTIVE /DES_SYSTEM/DES_MAP/LOCAL_DATA_BUS (value = X"04040000")
```

Another *Get Event* opcode was sent to the DES for LP1 to test the RAM update function because the first *Get Event* opcode for LP1 had a time tag of zero. The

data trace below shows the simulation time equal to five. LP1's RAM partition is shown below to validate the simulation time change. The second vector represents the simulation time which has been updated to five. This test validates the operation of the *Get Event* routine with the RAM device. All of the data traces for the *Get Event* opcode validate operation between the microcode and the hardware.

```
336307 NS
 M1: ACTIVE /DES_SYSTEM/DES_MAP/U2/RAM_RW/MEM_NIBBLE (value =
 LP1's RAM Partition:  X"00000004", X"00000005", X"00030001",
                       X"00000400", X"00000400", X"00001000",
                       X"00001400")
```

4. Post Event

The *Post Event* routine was thoroughly tested by monitoring the local and system bus for null messages. The count of operands that is contained in the lower order 10 bits of the null message and the time tag were examined for accuracy. The count should equal "1" whenever the event is a null message. The following data trace shows the null message retrieved from the DES and the associated count. From Table 10, the first value of the local data bus shows opcode formatted for a source and destination LP of one. The second vector represents the time tag which is zero for this event. The time tag was validated by reviewing the trace from the test data converted from the Intel Hypercube runs.

```
212749 NS
  M3:  ACTIVE /DES_SYSTEM/DES_MAP/LOCAL_DATA_BUS (value = X"10040401")
213349 NS
  M3:  ACTIVE /DES_SYSTEM/DES_MAP/LOCAL_DATA_BUS (value = X"00000000")
```

6.6 Summary

Even though a high-level test bench was implemented to interface with the DES coprocessor, the signals used were realistic signals that provide sufficient validation for the design. The test vectors were extracted from actual runs on the Intel Hypercube and

the test results were extracted from many DES simulations. All of these factors together validate the test process; therefore, the DES coprocessor works correctly and supports the Chandy-Misra protocol with null messages.

VII. Results and Recommendations

7.1 Introduction

A structural VHDL description of a DES simulation accelerator coprocessor was implemented to provide a proof of concept for simulation coprocessors. Taylor's requirements analysis provided the target areas for communications overhead reduction [21]. The CARWASH model was used to provide a general-purpose simulation for speedup determination. The SPECTRUM testbed filters were the communications tasks targeted for enhancement.

This chapter details the results and recommendations of this research effort. The calculation process used to obtain the speedup results are included in this chapter. An example calculation is provided to validate the calculation process. Additional areas to increase coprocessor performance are also outlined as part of the recommendations.

7.2 Calculation Process

Simulation speedup was calculated to quantify the results of this thesis effort. Without realistic event processing, the potential for speedup would be overstated; therefore, spin loops of 0, 1,000, and 100,000 were used to model an event being processed but cannot relate to true event processing times. The amount of speedup is application-dependent. If the time required for event processing is low, then the potential for speedup will be high. If event processing takes a considerable portion of the host's processing time, then the potential speedup decreases rapidly. The calculation process for determining overall system speedup followed the steps in Table 13. To ensure true speedup is stated, the average times for each routine are compared to the average hardware results. The average times do not predict peak performance speedup potential, but do provide reasonable speedup ratios.

7.2.1 Hypercube Filter Averages Simulation data was gathered from many simulation runs on the Intel Hypercube to provide sufficient filter information to average filter processing times. The simulation test data gathered from the Intel Hypercube provides accurate results for the four SPECTRUM filters. Figure 15 shows an example data segment extracted from a run with the spin loop set to zero.

Table 13. Speedup Procedures

Step	Speedup Procedure
1	Calculate Hypercube Filter Averages
2	Calculate DES Filter Averages
3	Calculate System Overhead
4	Final Speedup Calculation

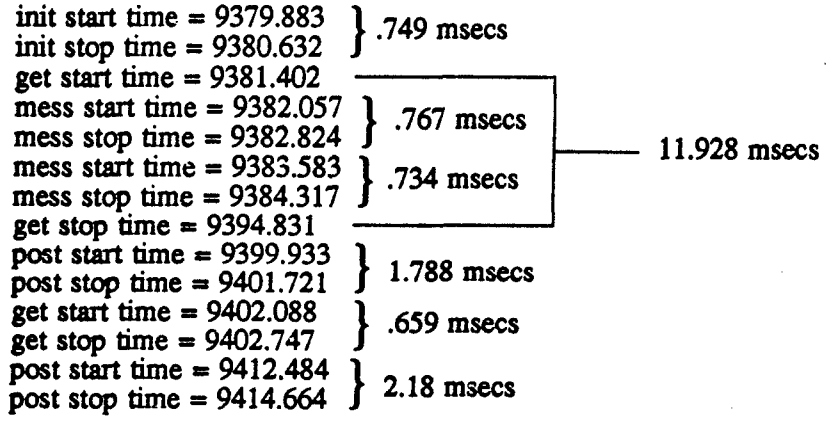

Figure 15. Hypercube Simulation Data

Table 14. Cube Filter Times

Filter	μ (msec)	Min (msec)	Max (msec)
Init Sim	.730	.652	.754
Post Msg	.808	.063	3.937
Get Event	9.405	.058	33.386
Get Modified	6.460	.058	30.118
Post Event	.708	.061	3.960

Table 15. DES Microcode Routine Test Data Processing Times

Filter	μ (msec)	Min (msec)	Max (msec)
Init Sim	.00741	.00584	.00900
Post Msg	.00272	.00216	.00312
Get Event	.00410	.00200	.00620
Post Event	.00401	.00392	.00428

Table 14 provides the average processing times and the percentage of overall processing time per filter for the Intel Hypercube iPSC/2. The *Get Modified* input was calculated by subtracting all of the filter calls made while an LP was blocking. The filter call should only be counted once to provide an accurate description. The value, 11.928 msecs, calculated for the first *Get Event* call in Figure 15 shows an example of filter calls being made during the *Get Event* filter call.

7.2.2 DES Filter Averages The simulation data was converted into opcodes and operands that could be understood by the DES. Assertion statements were inserted into the VHDL code to signal the start of each opcode. Simulation runs on the DES were conducted using the opcodes and operands to obtain sufficient data to calculate filter averages using the DES coprocessor. The processing times for the respective DES routines are included in Table 15.

7.2.3 System Overhead Calculation The system overhead provides the last piece of information required prior to calculating the total simulation speedup. The system

Hyprrcube / DES Mean Times

Wall Time = 19.795
Init Total Time = .749 } Init Mean = .749 / .00741
Init Calls = 1
Get Total Time = 12.587 } Get Event Mean = 6.294 / .00410
Get Calls = 2
Post Total Time = 3.968 } Post Event Mean = 1.984 / .00401
Post Calls = 2
Post Message Total Time = 1.501 } Post Message Mean = .7505 / .00272
Posrt Message Calls = 2

Wall Time - Total Filter Time = System Overhead

19.795 - 18.805 = .99 msec

Figure 16. Hypercube Total Times

overhead will be approximately the same with or without the DES coprocessor in use. The overhead had to be obtained for each of the three spin loops used to model event processing. Equation 1 was used to compute the system overhead for each of the spin loops.

$$System\ Overhead = simulation\ wall\ time - filter\ processing\ time \quad (1)$$

An example calculation of system overhead is shown in Figure 16. To provide more realistic filter processing percentages, the data required for this step was collected without the print statements in the code. The overhead calculated for each of the spin loops is included in Table 16.

Filter speedup was also calculated to ensure the microcode implementation is an approach worthy of consideration. The results show that the microcode implementation appears to be reasonable. Equation 2 was used to compute the filter speedup and the results are included in Table 17.

Table 16. System Overhead

Spin loop	Overhead (msec)
0	(.295 - .2908) = .00472
1,000	(.308 - .280588) = .02741
100,000	(4.038 - .904512) = 3.1335

Table 17. Coprocessor Speedup Ratios

Filter	Filter Speedup
Init Sim	98.5
Post Msg	297.1
Get Event	1575.6
Post Event	176.6

$$FilterSpeedup = (CUBE\_TIMES)/(DES\_TIMES) \quad (2)$$

7.2.4 Overall Speedup The CARWASH simulation was executed with spin loops to emulate the event processing times. This information provides a speedup range depending on the application. Table 18 provides a detailed summary of the DES coprocessor percentage of processing dedicated to filter execution and the speedup obtained for each spin loop. The final speedup results were calculated by finding the total simulation time for the cube divided by the total time for the DES coprocessor. Equation 3 shows the formula used to calculate speedup.

$$Speedup = (Simulation\ Time)/((\Sigma\ (Filter\ x\ Filter\ Calls)) + Overhead) \quad (3)$$

Table 18. Overall Speedup using Spin Loops

Spin Loop	Filter Time as % of Total Processing	Speedup
0	98.4	60.32
1,000	91.1	11.16
100,000	22.4	1.29

7.3 Recommendations

Several areas concerning the CAM, microcode, and DES coprocessor in general were revealed during this research effort. Some were explored and added to the general-purpose hardware coprocessor design. The following subsections review the areas to be further examined to potentially provide greater speedup.

7.3.1 CAM Modifications The CAM used within the DES architecture has been modified to provide a maxima and minima for a subset of CAM words. This modification will provide additional speedup for the *Get Event* routine. This modification eliminates the hardware implementation problems discussed in Chapter IV. The problem occurs when many words attempt to raise a line high and only one drives the line low. This pull-down capability is not realistic and has been resolved in the new CAM. Only a few modifications to the front-end driver will be required to provide this capability.

7.3.2 Microcode Enhancements in the microcode are always possible. A more detailed look at the microcode implementation should be approached to ensure maximum performance. RAM usage as well as microcode efficiency should be researched to provide maximum speedup.

The present architecture is a decimal approach to instruction translation. All of the control store addresses, instructions, registers, and JUMP addresses are read from a file in a decimal format and then translated into a binary format by the test bench. An assembler should be designed to translate the microcode instructions into the binary format required by the DES. The program should be a multi-pass assembler allowing the use of labels for

JUMP addresses. The instruction addresses should automatically be generated to reduce user overhead.

7.3.3 Behavioral Components Approximately 90 percent of the components have been converted to a gate-level structural VHDL format. The entire design should be converted and tested thoroughly. MAGIC layouts have been completed for SRAM and CAM devices within AFIT. Both of these devices should be tested thoroughly for compatibility. Prior to a MAGIC layout being attempted, all of the VHDL structural components should be at the gate level.

7.3.4 Timing Analysis A critical timing analysis should be accomplished to obtain peak performance at all times. Each phase in the four-phase clock is presently set to 10 ns. The critical units should be obtained to enable minimal phase widths. If each phase can be reduced, the potential for additional speedup can be increased. Once an HSPICE timing analysis has been conducted on each component the time delays should be updated in the structural descriptions.

7.3.5 Paradigm Support An analysis of other paradigms should be conducted to ensure the DES coprocessor is general purpose enough to support various algorithms. Variations of the Chandy-Misra protocol should be decomposed to ensure DES usability. The optimistic Time-Warp protocol also seems to be natural selection to be coded to work on the DES.

7.3.6 Hardware Implementation The DES coprocessor should be implemented on the Intel Hypercube iPSC/2. The coprocessor would require a significant redesign to be implemented on the iPSC/1. The DES provides a 32-bit bus for opcode and operand transfer. The iPSC/1 only provides a 16-bit bus. The interfacing issues should be confronted early in the next thesis cycle.

7.4 *Summary*

The DES coprocessor was designed with general-purpose simulation support as the primary design objective. The microcode was written to support the Chandy-Misra protocol with null messages. A test bench was then designed to effectively test the interrupt and routines, as well as opcode and operand execution.

The speedup varies from 60.32 to 1.29 times when using the DES coprocessor. These results are more promising for fine-grained (spin loop = 0) than coarse-grain (spin loop = 100,000) applications. In fine-grained applications, the DES coprocessor is promising because the synchronization overhead will no longer be a bottleneck. In coarse-grained applications, the DES coprocessor is not as promising because the event processing will be the bottleneck.

Appendix A. *DES SPECTRUM Algorithms*

The following SPECTRUM algorithms are followed to directly support the Chandy-Misra paradigm. The algorithms implement the corresponding filters used in SPECTRUM. The code drives the control driven architecture.

A.1 Read-Only Control Store Procedure

```
This algorithm is designed to load the control store and
mapping RAM.  System address bit two is a zero if the data is
an opcode and a one if the data is an operand.  An operand
for this routine is composed of microinstructions for the
control store or mapped addresses for the mapping RAM.

    1.  Initialize CAM
    2.  Signal ready to the host
    3.  When OPCODE, check for = 0
          - if equal 0 goto step 4
          - else goto step 3
    4.  Wait for data present
          - if present then goto 5
          - else LOOP (GOTO step 4)
    5.  Check to see if it is an OPCODE
          - if OPCODE then goto step 8
          - else continue (MUST BE AN OPERAND)
    6.  Load data into the control store or mapping RAM
    7.  JUMP to step 4
    8.  Check OPCODE = 0
          - if equal 0 then goto START_OF_FETCH_DECODE
          - else SIGNAL_ERROR(RELOAD_DATA) and goto step 3
```

A.2 Fetch/Decode Procedure

```
This algorithm details the operation of the fetch/decode routine.
This routine loads the common registers for future use and calls
the appropriate function.  Register 63 is loaded with the count
of operands to follow the opcode, register 27 is loaded with the
TOLP information, and register 57 (accumulator) is loaded with
```

the entire opcode for use when selecting the base pointers or status registers for the specified LP. This operation will be explained in the actual microcode. Register 56 is the instruction register and also contains the entire opcode. Register 56 is used to load the other registers.

1. Wait for data present
   - if no data present then LOOP (goto step 1)
2. Check for OPCODE
   - if OPCODE then goto step 3
   - else
     -- SIGNAL_ERROR(BAD_OPCODE) using Reg22
        --- data = 00000000000000000000000111111111
     -- remove data from the PARIO
     -- reset WRITE_REMOTE/READ_LOCAL status bit
     -- goto step 1
3. Read data into IR
4. Load count into register 63
5. Load TO_LP into register 27
6. Load IR into the ACCumulator
7. JUMP to IR address

A.3 Initialize Simulation Procedures

In general, this algorithm setup the LP specific information in RAM, setup the status registers for the specified LP, reserve words in the CAM for every input arc, and output a null message to every output arc. Register 23 will be loaded with the LP delay, register 62 will be loaded with the simulation time, and register 21 will be loaded with the number of I/O arcs. These registers are throughout the intialize simulation routine.

1. Wait for data present
   -- if no data present then LOOP (goto 1)
2. Check for OPCODE
   -- if OPCODE goto 27 ***UNLOAD PARIO AND RESTART***
   -- else start load of simulation data
3. Load LP_DELAY into register 23
4. Reset READ_LOCAL/WRITE_REMOTE status bit
5. Add -1 to count (Reg63) and check = 0
   -- if count = 0 then goto 25 ***ERROR***

```
          ---  SIGNAL ERROR(BAD_INIT) goto FETCH_DECODE
          ---  data = 00000000000000000000000011111111
 6.  Wait for data present
     --  if no data present then LOOP (goto 6)
 7.  Check for OPCODE
     --  if OPCODE goto 27 ***UNLOAD PARIO AND RESTART***
 8.  Load SIM_TIME into register 62
 9.  Reset READ_LOCAL/WRITE_REMOTE status bit
10.  Add -1 to count (Reg63) and check = 0
     --  if count = 0 then goto 25 ***ERROR***
          ---  SIGNAL ERROR(BAD_INIT) goto FETCH_DECODE
          ---  data = 00000000000000000000000011111111
11.  Wait for data present
     --  if no data present then LOOP (goto 11)
12.  Check for OPCODE
     --  if OPCODE goto 27 ***UNLOAD PARIO AND RESTART****
13.  Load #_ARCS_IN/OUT into register 21
14.  Reset READ_LOCAL/WRITE_REMOTE status bit
15   Add -1 to count (Reg63) and check = 0
     --  if count = 0 then goto 25 ***ERROR***
          ---  SIGNAL ERROR(BAD_INIT) goto FETCH_DECODE
          ---  data = 00000000000000000000000011111111
16.  Load #_ARCS_IN in register 29
17.  Store base pointer in register 31
18.  Store LP_DELAY, SIM_TIME, and #_ARCS in RAM
19.  Setup ARCS_STATUS_REGISTER
20.  Write ARCS_IN directly to RAM
21.  Write ARCS_OUT directly to RAM
22.  Check count = 0
     --  if count /= 0 SIGNAL_ERROR(INIT_DONE) goto 23
     --  data = 00000000000000000000000001111111
23.  Save one word in partitioned CAM for each INPUT_ARC
24.  JUMP to FETCH_DECODE
25.  SIGNAL_ERROR(BAD_INIT)
26.  JUMP to 24
27.  Unload PARIO
28.  Reset READ_LOCAL/WRITE_REMOTE
29.  JUMP to 1
```

A.4 Post Message Procedures

In general, this algorithm loads the event into the CAM and adjacent RAM, and updates the status register for the specified LP. Register 26 is used to store the time tag and register 30 is used to store the memory pointer.

1. Wait for data present
   - if no data present then LOOP (goto 1)
2. Load TIME_TAG in register 26
3. Add -1 to count and check = 0
   - if count = 0 then goto step 10 ***NULL MESSAGE LOAD***
4. Wait for data present
   - if no data present then LOOP (goto 4)
5. Load MEM_PTR in register 30
6. Write to partitioned CAM if free;
7. Check CAM_FULL status
   - if FULL then SIGNAL_INTR(DATA);
     -- data = 01111111
8. Update ARCS_IN_STATUS register
9. JUMP to Fetch/Decode
10. Load register 30 with all 0's
11. JUMP to step 6

A.5 Get Event Procedures

In general, this algorithm checks to see if an event is ready, retrieves the event, sends it to the host processor, and updates the status register for the specified LP. If a null message is retrieved, nulls are sent to all output arcs a the process starts over. In order to support the Chandy-Misra paradigm, nulls have to be sent to all output arcs when a null is retrieved.

1. Check to see if event ready
2. Find minima for specified LP
   - this word is the next event
3. Retrieve MEM_PTR
4. Update SIM_TIME for specified LP
5. Check for NULL
   - if NULL goto 10
6. Reformat CAM word for transfer
7. Output to CUBE
8. Update LP STATUS register

- check CAM for another EVENT for specified ARC
- update accordingly

9. JUMP to FETCH/DECODE
10. Send NULL messages to all OUTPUT arcs
11. Update LP STATUS register
    - check CAM for another EVENT for specified ARC
    - update accordingly
12. Output NULL messages to all output arcs
13. JUMP to 1

A.6 Post Event Procedures

This algorithm sends null messages to all output arcs other than the arc specified in the opcode.

1. Store ARC info from message in a register
2. Retrieve RAM ptr for specified TO_LP
3. Obtain number of Input Arcs
4. Obtain number of Output Arcs
5. Advance pointer to first Output Arc
6. Read an Output Arc
7. If equals Arc from message
   - then goto step 10
8. Format Output message
9. Interrupt CUBE and send message
10. Advance PTR
11. Add -1 to #_OUT_ARCS
12. Check for equal zero
    - if equal zero then JUMP to FETCH/DECODE
13. JUMP to step 6

Appendix B. *DES Microcode Routines*

The following microcode was implemented according to the algorithms in Appendix A. The available commands are listed in Appendix C.

B.1 Read-Only Microcode

This section of code is required to load the control store and the mapping ram whenever the DES is reset.

```
*************************************************************
**
**  Lines 0 - 3 are house cleaning instructions.  These
**  commands prepare the CAM and DES for processing.
**
*************************************************************

 0.  JUMP to 1;

 1.  CAM_INIT;

     -  this instruction initializes the CAM for use

 2.  If not (CAM_COMPLETE) goto 2

     -  loop until initialization complete

 3.  SIGNAL_READY;

     -  set status to READY

*************************************************************
**
**  Lines 4 - 9 are used to read the opcode and check to see
**  if it equals zero.  If the opcode equals zero, then
**  initialization can begin.  Else the DES will wait for
**  the next opcode.
**
*************************************************************
```

```
4.  If not (OPCODE and READ_LOCAL) goto 4

    -  wait for data present
    -  an opcode is expected to start the load

5.  Input_Data;

6.  Reg56 := MBR;

7.  If ZERO goto 10

8.  Read_Local_Toggle;

    -  toggle read_local bit of the status register

9.  JUMP to 4;

*****************************************************************
**
**  Lines 10 - 15 read microinstructions into the control
**  store.  This code loops until an opcode is encountered.
**
*****************************************************************

10.  BEGIN_LOAD;

     -  change control store state to load

11.  If not (READ_LOCAL) goto 11

12.  If OPCODE goto 16

13.  Input_Data;

14.  Read_Local_Toggle;

15.  JUMP to 11;

*****************************************************************
**
**  Lines 16 - 19 are required to read the opcode, check to
**  see if it equals zero, and jump accordingly.  If the
**  opcode equals zero then the loading has completed in a
**  correct manner.  If the opcode does not equal zero, then
```

```
**  the load i     be restarted.
**
*****************************************************************

16.  Input_Data;

17.  Read_Local_Toggle;

18.  Reg56 := MBR;

     -  place contents of MBR in register 56

19.  If ZERO goto 23

*****************************************************************
**
**  Lines 20 - 22 are required to signal an error and jump
**  to the address 4 to restart the initial load.
**
*****************************************************************

20.  MBR := Reg60;

21.  SIGNAL_ERROR;

     -  toggle the error bit and write_local bit in the
     status register

22.  JUMP to 4;

*****************************************************************
**
**  Lines 23 and 24 are used to end the successful load and
**  jump to the fetch/decode routine.
**
*****************************************************************

23.  END_LOAD;

24.  JUMP to 32;
```

B.2 Fetch/Decode Microcode

This routine is designed to load all common registers and call the correct routine. The TOLP, COUNT, accumulator, and IR are all loaded with the appropriate data for processing.

```
*****************************************************************
**
**  Lines 32 and 33 wait for data and then ensures it is an
**  opcode before continuing.
**
*****************************************************************

 32.  If not (READ_LOCAL) then goto 32

      -  Wait for data

 33.  If OPCODE then goto 41

      -  Check for OPCODE

*****************************************************************
**
**  Lines 34 - 40 compose a routine that is called whenever
**  an operand is read when an opcode should have been read.
**  The error vector = "0111111111" for this type of error.
**
*****************************************************************

 34.  Reg22 := BAND(Reg22, 0);

 35.  Reg22 := BOR(Reg22, Reg55);

 36.  Reg22 := RSHIFT8(Reg22);

 37.  Reg22 := RSHIFT8(Reg22);

 38.  Output_Data;

 39.  SIGNAL_ERROR;

 40.  JUMP to 32
```

```
************************************************************
**
**  Lines 41 - 51 are required to load the count in reg63,
**  TOLP info in reg27, opcode in reg56 (IR), and reg57.
**  Register 57 is the accumulator.  Masks are used to
**  ensure only the desired data is loaded in the target
**  register.
**
************************************************************

 41.  Reg63 := BAND(Reg63, Reg52);

 42.  Reg27 := BAND(Reg27, Reg52);

 43.  Reg57 := LSHIFT(BAND(Reg57, R2_MUX(ACC, 0)));

 44.  Reg63 := BOR(Reg63, Reg60);

 45.  Reg27 := BOR(Reg27, Reg59);

 46.  Input_Data;

 47.  Reg56 := MBR;

 48.  Reg63 := BAND(Reg63, Reg56);

 49.  Reg27 := BAND(Reg27, Reg56);

 50.  Reg57 := BOR(Reg57, Reg56);

 51.  Read_Local_Toggle;

************************************************************
**
**  Lines 52 and 53 are required to signal to the node that
**  the DES is ready and then to jump to the code specified
**  in the mapping ram for the routine identified in the IR.
**
************************************************************

 52.  SIGNAL_READY;

 53.  JUMP to Mapping_RAM(IR);
```

```
*************************************************************
**
**  Lines 500 and 501 are used to set the DES back into a
**  ready state.  Processing will not continue until the DES
**  has returned to the ready state.
**
*************************************************************

 54.  SIGNAL_READY;

 55.  JUMP to 32;
```

B.3 Initialize Simulation Microcode

This code is used to load all LP specific information into DES RAM, reserve words in the CAM for all input arcs, setup the status register for each LP, and output null messages to all output arcs. This routine has to be executed for each LP in a given simulation.

```
*************************************************************
**
**  Lines 60 and 61 will force the DES to wait for data,
**  check to see if it is an opcode (Only operands should
**  be sent to the DES at this time), and jump to the error
**  routine if an operand is read.
**
*************************************************************

 60.  If not (READ_LOCAL) then goto 60

     -  wait for data present, loop if not

 61.  If OPCODE then goto 175

     -  should be an operand

*************************************************************
**
**  Lines 62 - 66 load the LP_DELAY for the specified LP
**  into register 23, reset the read_local bit of the status
**  register, decrement and check the count register, and
```

```
**  jump to line 175 if the count equals zero.
**  NOTE:  The count should not equal zero because the
**  simulation time and arc information has not been read.
**
*****************************************************************

 62.  Input_Data;

     -  this command will enable data onto the local data bus

 63.  Reg23 := MBR;

     -  Load the LP_DELAY into register 23

 64.  Reg63 := Reg63 + Reg54;

     -  decrement count

 65.  If ZERO then goto 175

     -  count should not be zero yet;  still have to load
     SIM_TIME, #_ARCS and the I/O_ARCS

 66.  Read_Local_Toggle;

*****************************************************************
**
**  Lines 67 - 73 load the simulation time for the LP
**  into register 62, reset the read_local bit of the status
**  register, decrement and check the count register, and
**  jump to line 175 if the count equals zero.
**  NOTE:  The count should not equal zero because the arc
**  information has not been read.
**
*****************************************************************

 67.  If not (Read_Local) goto 67

 68.  If OPCODE then goto 175

     -  should be an operand

 69.  Input_Data;

     -  this command will enable data onto the local data bus
```

```
70.  Reg62 := MBR;

     -  Load the SIM_TIME into register 62

71.  Reg63 := Reg63 + Reg54;

     -  decrement count

72.  If ZERO then goto 175

73.  Read_Local_Toggle;

*****************************************************************
**
**  Lines 74 - 80 load the number of I/O arcs for the LP
**  into register 21, reset the read_local bit of the status
**  register, decrement and check the count register, and
**  jump to line 175 if the count equals zero.
**  NOTE:  The count should not equal zero because all of
**  the arc information has not been read.
**
*****************************************************************

74.  If not (READ_LOCAL/WRITE_REMOTE) then goto 74

     -  wait for data present, loop if not

75.  If OPCODE then goto 175

     -  should be an operand

76.  Input_Data;

     -  this command will enable data onto the local data bus

77.  Read_Local_Toggle;

78.  Reg21 := MBR;

     -  Load the #_ARCS_IN/OUT into register 21

79.  Reg63 := Reg63 + Reg54;

     -  decrement count
```

```
 80.  If ZERO then goto 175

      -  count should not be zero yet;  the I/O_ARCS

*****************************************************************
**
**  Lines 81 and 82 load the base pointer for the LP into
**  register 31 for use when loading the input and output
**  arcs into RAM.  R2_MUX(ACC, 0) specifies the LP base
**  pointer.  The '0' is used for bit5 of the R2_MUX to
**  point to registers 0 through 19.  A '1' would be used
**  to specify registers 32 through 51 (status registers).
**
*****************************************************************

 81.  Reg31 := BAND(Reg31, Reg52);

      -  zero register 31

 82.  Reg31 := BOR(Reg31, R2_MUX(ACC, 0));

      -  Load the base pointer into register 31

*****************************************************************
**
**  Lines 83 - 91 are responsible for storing the delay for
**  the LP, simulation time, and the number of I/O arcs into
**  the LP's RAM partition.
**
*****************************************************************

 83.  MBR := Reg23; MAR := Reg31;

      -  start store of the LP_DELAY into DES RAM

 84.  RAM_WRITE;

      -  causes write to RAM

 85.  Reg31 := Reg31 + Reg53;

      -  advance RAM ptr

 86.  MBR := Reg62;  MAR := Reg31;
```

```
    - start store of the SIM_TIME into DES RAM

 87.  RAM_WRITE;

    - causes write to RAM

 88.  Reg31 := Reg31 + Reg53;

    - advance RAM ptr

 89.  MBR := Reg21;  MAR := Reg31;

    - start store of the #_ARCS into DES RAM

 90.  RAM_WRITE;

    - causes write to RAM

 91.  Reg31 := Reg31 + Reg53;

    - advance RAM ptr

*****************************************************************
**
**  Lines 92 - 94 sets up register 29 to be used when
**  setting up the status register.  Register 29 will
**  contain the number of I/O arcs after these instructions.
**
*****************************************************************

 92.  Reg29 := BAND(Reg29, Reg52);

    - zero register 29

 93.  Reg29 := BOR(Reg29 , Reg60);

    - prepare register 29 to AND with register 21 to obtain the
#_ARCS_IN

 94.  Reg29 := BAND(Reg29, Reg21);

    - register 29 now contains the #_ARCS_IN

*****************************************************************
```

```
**
**  Lines 95 - 101 sets up the status register for the
**  specified LP.  The R1_MUX(ACC, 1) is used to select the
**  the status register instead of the base pointer.
**
*****************************************************************

 95.  R1_MUX(ACC, 1) := BAND(R1_MUX(ACC, 1), Reg52);

      - zero STATUS register for specified LP
      - the '1' selects the STATUS register

 96.  R1_MUX(ACC, 1) := LSHIFT(R1_MUX(ACC, 1) + Reg53);

      - preparing STATUS register

 97.  Reg29 := Reg29 + Reg54;

      - decrement ARCS_IN_COUNTER

 98.  If ZERO then goto 100

 99.  JUMP to 96;

100.  R1_MUX(ACC, 1) := RSHIFT(R1_MUX(ACC, 1));

101.  R1_MUX(ACC, 1) := BXOR(R1_MUX(ACC, 1), Reg60);

      Reg 60     00000000000000000000000000001111111111
      Stat Reg.  00000000000000000000000000000011111111
      Result     00000000000000000000000000001100000000

      - the status register is now ready
      - this is an example of an LP with 8 input arcs

*****************************************************************
**
**  Lines 102 - 104 load the number of input arcs into
**  register 29 for input arc loading.
**
*****************************************************************

102.  Reg29 := BAND(Reg29, Reg52);

      - zero register 29 again
```

```
103.  Reg29 := BOR(Reg29, Reg60);

104.  Reg29 := BAND(Reg29, Reg21);

      -  register 29 now contains ARCS_IN count

*****************************************************************
**
**  Lines 105 - 126 compose a loop which loads the input
**  arcs into RAM and then reserves a word in CAM for
**  future use.  The number of input arcs is decremented
**  and checked each time to determine when the loop has
**  completed.  The count is also checked each time.
**
*****************************************************************

105.  If not (Read_Local) goto 105

106.  If OPCODE goto 175

107.  Input_Data;

108.  Reg22 := MBR;

109.  MBR := Reg22;  MAR := Reg31;

110.  RAM_WRITE;

111.  Read_Local_Toggle;

112.  Reg24 := BAND(Reg24, Reg52);

113.  Reg24 := BOR(Reg24, Reg55);

114.  Reg24 := BAND(Reg24, Reg22);

      -  store FROM_NODE/LP in register 24

115.  Reg26 := BAND(Reg26, Reg52);

116.  Reg26 := BOR(Reg26, Reg59);

117.  Reg26 := BAND(Reg26, Reg27);
```

```
      -  store TO_NODE/LP in register 26

118.  Reg26 := LSHIFT8(BOR(Reg26, Reg24));

119.  MBR := Reg26;

120.  CAM_RESERVE_ARC;

121.  Reg31 := Reg31 + Reg53;

122.  Reg29 := Reg29 + Reg54;

123.  If ZERO then goto  127

124.  Reg63 := Reg63 + Reg54;

125.  If ZERO goto 175

126.  JUMP to 105;

127.  Reg63 := Reg63 + Reg54;

128.  If ZERO then goto 175

*****************************************************************
**
**  Lines 129 - 144 compose a loop which loads the output
**  arcs into RAM.  The count is checked each time to
**  determine when the loop is completed.  Register 29 is
**  loaded again using register 21 and right shifted 16
**  times to obtain the number of output arcs which is
**  located in the leftmost 16 bits of the word.  After
**  the first time through the loop the return address is
**  133 because the number of output arcs does not need to
**  be recomputed.
**
*****************************************************************

129.  Reg29 := BAND(Reg29, Reg52);

      -  zero register 29 to use for counter for OUTPUT arcs

130.  Reg29 := RSHIFT8(BOR(Reg29, Reg21));

131.  Reg29 := RSHIFT8(Reg29);
```

```
132.  Reg25 := BOR(Reg25, Reg29);

133.  If not (Read_Local) goto 133

134.  If OPCODE goto 175

135.  Input_Data;

      -  this command will enable data onto the local data bus

136.  Reg22 := MBR;

137.  MBR := Reg22;  MAR := Reg31;

138.  RAM_WRITE;

139.  MAR := Reg31;

140.  Reg29 := Reg29 + Reg54;

141.  If ZERO then goto 175

142.  Reg63 := Reg63 + Reg54;

143.  If ZERO then goto 175

144.  JUMP to 133;

*****************************************************************
**
**  Lines 145 - 155 compose a set of commands that are used
**  to setup the registers and the RAM base pointer for use
**  when formatting and transferring null messages to start
**  the simulation.
**
*****************************************************************

145.  Reg31 := BAND(Reg31, Reg52);

146.  Reg31 := BOR(Reg31, R2_MUX(ACC, 0));

      -  reset the base pointer to start of partition

147.  Reg31 := Reg31 + Reg53;
```

```
148.  Reg29 := BAND(Reg29, Reg52);

149.  Reg31 := Reg31 + Reg53;

150.  Reg29 := BOR(Reg29, Reg60);

151.  Reg29 := BAND(Reg29, Reg21);

152.  Reg27 := RSHIFT8(Reg27);

153.  Reg31 := Reg31 + Reg53;

154.  Reg31 := Reg31 + Reg29;

      - advance pointer to start of ARCS_OUT

155.  Reg23 := Reg23 + Reg62;

      - register 23 now contains the TIME_TAG

*************************************************************
**
**  Lines 156 - 174 are used to complete data packet
**  formatting and sending the null messages to the node
**  processor.  Line 161 inserts a '1' in the lowest order
**  bit to specify that there will be one operand following
**  the original data packet.  The information following
**  will be the time tag for the message.
**
*************************************************************

156.  MAR := Reg31;

157.  RAM_READ;

158.  Reg24 := MBR;

159.  Reg24 := LSHIFT8(Reg24);

      - this command shifts the OUTPUT_NODE/LP into the
      TO_NODE/LP field for the POST EVENT message

160.  Reg24 := BOR(Reg24, Reg27);
```

```
161.  Reg24 := BOR(Reg24, Reg53);

162.  If not (Write_Local) then goto 162

163.  MBR := Reg24;

      -  all 1's => POST EVENT Interrupt

164.  Output_Data;

165.  MBR := Reg60;

166.  SIGNAL_INTERRUPT;

167.  If not (Write_Local) goto 167

168.  MBR := Reg23;

169.  Output_Data;

      - place data in the PARIO device

170.  Write_Local_Toggle;

171.  Reg31 := Reg31 + Reg53;

172.  Reg29 := Reg29 + Reg54;

173.  If ZERO goto 500

174.  JUMP to 156;

**************************************************************
**
**  Lines 175 - 181 compose an error routine called whenever
**  an opcode is read.  Only operands should be read during
**  the initialize simulation routine.  The error vector for
**  this error is 1111111111.
**
**************************************************************

175.  If not (Write_Local) goto 175

176.  MBR := Reg60;
```

```
177.  Output_Data;

178.  Write_Local_Toggle;

179.  SIGNAL_ERROR;

180.  If not (Write_Local) goto 180

181.  JUMP to 54;
```

B.4 Post Message Microcode

```
**********************************************************
**
**  Lines 198 - 204 are written to wait for data, which is
**  the time tag, read it into register 26, toggle the
**  read_local bit of the status register, decrement the
**  counter, and check to see if count equals zero.
**
**********************************************************

198.  If not (READ_LOCAL) then goto 198

      - wait for data present

199.  If OPCODE then goto 260

200.  Input_Data;

      - this command will enable data onto the local data bus

201.  Read_Local_Toggle;

202.  Reg26 := MBR;

      - Load TIME_TAG into register 26

203.  Reg63 := Reg63 + Reg54;

      - decrement count
```

```
204.  If ZERO goto 268

*****************************************************************
**
**  Lines 205 - 209 are written to wait for data, which is
**  the memory pointer, read it into register 30, and toggle
**  the read_local bit of the status register.
**
*****************************************************************

205.  If not (Read_Local) goto 205

206.  If OPCODE goto 260

207.  Input_Data;

208.  Read_Local_Toggle;

209.  Reg30 := MBR;

     -  Load the memory pointer into register 30

*****************************************************************
**
**  Lines 210 - 217 places the TOLP/NODE information into
**  register 24 as part of the formatting routine to store
**  the word into the CAM.
**
*****************************************************************

210.  Reg24 := BAND(Reg24, Reg52);

211.  Reg24 := BOR(Reg24, Reg59);

212.  Reg24 := LSHIFT(BAND(Reg24, Reg27));

     -  TO_info is now located in register 24

213.  Reg24 := LSHIFT(Reg24 + Reg24);

     -  double left shift

214.  Reg24 := LSHIFT(BAND(Reg24, Reg59));

     -  removes the TO_NODE field from the register
```

```
215.  Reg24 := LSHIFT(Reg24 + Reg24);

      - double left shift

216.  Reg22 := BAND(Reg22, Reg52);

      - double left shift
      - register 24 now has the TO_LP field properly located

217.  Reg24 := LSHIFT(Reg24 + Reg24);

*****************************************************************
**
**  Lines 218 - 222 adds the FROM LP/NODE information and
**  time tag into register 24 as part of the formatting
**  routine to store the word into the CAM.
**
*****************************************************************

218.  Reg22 := BOR(Reg22, Reg55);

      - register 22 now contains the FROM field
      - it must be left shifted

219.  Reg22 := BAND(Reg22, Reg57);

220.  Reg22 := LSHIFT8(Reg22);

221.  Reg24 := BOR(Reg24, Reg22);

222.  Reg24 := BOR(Reg24, Reg26);

*****************************************************************
**
**  Lines 223 - 226 writes the event to the CAM.  The DES
**  does not continue processing until the CAM has signalled
**  back to the DES that the CAM is not full.  The CAM_MATCH
**  flag is used to determine if the CAM is full.  A jump to
**  address 270 means the CAM is full.
**
*****************************************************************

223.  MBR := Reg24;
```

```
224.  CAM_WRITE_WORD;

225.  If not (CAM_COMPLETE) goto 225

226.  If not (CAM_COMPLETE) goto 270

227.  Reg31 := BAND(Reg31, Reg52);

228.  Reg31 := BOR(Reg31, R2_MUX(ACC, 0));

      -  stores the base pointer for RAM in register 31

229.  Reg31 := Reg31 + Reg53;

*****************************************************************
**
**  Lines 230 and 231 writes the memory pointer to the
**  adjacent RAM.
**
*****************************************************************

230.  MBR := Reg30;

231.  ADJ_RAM_WRITE;

*****************************************************************
**
**  Lines 232 - 242 are responsible for preparing for
**  status register updating.  The base pointer has to be
**  advanced to the first input arc and the number of arcs
**  has to be retrieved for arc reading.
**
*****************************************************************

232.  Reg31 := Reg31 + Reg53;

233.  MAR := Reg31;

234.  RAM_READ;

235.  Reg30 := MBR;

      -  read number of input arcs into register 30

236.  Reg31 := Reg31 + Reg53;
```

```
237.  Reg20 := BAND(Reg20, Reg52);

238.  Reg20 := BOR(Reg20, Reg53);

239.  Reg22 := BAND(Reg22, Reg52);

240.  Reg22 := BOR(Reg22, Reg55);

241.  Reg30 := BAND(Reg30, Reg60);

      - this command loads the FROM field into register 22 for
      comparison to the RAM input arcs

242.  Reg22 := BAND(Reg22, Reg57);

*****************************************************************
**
**  Lines 243 - 252 compose a loop which determines which
**  arc a message has been received on and sets up a bit
**  pattern to be used when updating the status register.
**  Lines 253 and 254 performs the updating of the status
**  register.  A simple OR instruction is used to set the
**  appropriate bit to a 1.
**
*****************************************************************

243.  MAR := Reg31;

244.  RAM_READ;

245.  Reg25 := MBR;

246.  ALU := BXOR(Reg25, Reg22);

247.  If ZERO then goto 253

248.  Reg31 := Reg31 + Reg53;

249.  Reg20 := LSHIFT(Reg20);

250.  Reg30 := Reg30 + Reg54;

251.  If ZERO then goto 276
```

```
252.  JUMP to 243;

253.  R1_MUX(ACC, 1) := BOR(R1_MUX(ACC, 1), Reg20);

254.  JUMP to 500;

*****************************************************************
**
**  Lines 260 - 266 compose the routine which specifies an
**  error has occurred.  The error message sent to the host
**  processor signifies that an opcode was received when an
**  operand was expected.
**
*****************************************************************

260.  If not (Write_Local) goto 260

261.  MBR := Reg60;

262.  Output_Data;

263.  Write_Local_Toggle;

264.  Signal_Error;

265.  If not (Write_Local) goto 265

266.  JUMP to 500;

*****************************************************************
**
**  Lines 268 and 269 are used to load a null message into
**  register 30.  These two lines of code are called from
**  whenever a message is received an no memory pointer is
**  specified.
**
*****************************************************************

268.  Reg30 := BAND(Reg30, Reg52);

269.  JUMP to 210;

*****************************************************************
**
**  Lines 270 - 276 compose an error routine which is called
```

```
**  whenever the CAM is full.  The error vector = 00000001.
**
*****************************************************************

270.  If not (Write_Local) goto 270

271.  MBR := Reg53;

272.  Output_Data;

273.  Write_Local_Toggle;

274.  SIGNAL_ERROR;

275.  If not (Write_Local) goto 275

276.  JUMP to 54;

*****************************************************************
**
**  Lines 277 - 287 compose an error routine which is called
**  whenever there are no matching arcs for the destination
**  LP.  The error vector = 11111111.
**
*****************************************************************

277.  Reg30 := BAND(Reg30, Reg52);

278.  Reg30 := BOR(Reg30, Reg60);

279.  Reg30 := RSHIFT(Reg30);

280.  Reg30 := RSHIFT(Reg30);

281.  If not (Write_Local) goto 281

282.  MBR := Reg30;

283.  Output_Data;

284.  Write_Local_Toggle;

285.  Signal_Error;

286.  If not (Write_Local) goto 286
```

```
287.  JUMP to 500;
```

B.5 Get Event Microcode

```
*************************************************************
**
**  Lines 349 - 352 checks to see if an event is ready for
**  the specified LP.
**
*************************************************************

349.  Reg22 := BAND(Reg22, Reg52);

350.  Reg22 := BOR(Reg22, R2_MUX(ACC, 1));

      -  store STATUS register for specified LP in register 22

351.  ALU := BXOR(Reg22, Reg60);

352.  If ZERC then goto 362

*************************************************************
**
**  Lines 353 - 361 compose an error routine which signals
**  the host processor that an event is not ready.
**
*************************************************************

353.  Reg22 := BAND(Reg22, Reg52);

354.  Reg22 := RSHIFT(BOR(Reg22, Reg60));

355.  Reg22 := RSHIFT(Reg22);

356.  MBR := Reg22;

357.  Output_Data;

358.  SIGNAL_ERROR;
```

```
359.  Write_Local_Toggle;

360.  If not (Write_Local) goto 360

361.  JUMP to 54;

*****************************************************************
**
**  Lines 362 - 364 are used to format a 32-bit message to
**  be used by the CAM's front end driver to locate the
**  event with the smallest time tag for the appropriate LP.
**
*****************************************************************

362.  Reg24 := BAND(Reg24, Reg52);

363.  Reg24 := BOR(Reg24, Reg27);

364.  Reg24 := LSHIFT8(Reg24);

*****************************************************************
**
**  Lines 365 - 368 commands the CAM to perform a search for
**  the minimum time tag for the specified LP.  The DES will
**  wait until the CAM has returned control to the DES.
**
*****************************************************************

365.  MBR := Reg24;

366.  CAM_MIN_FIND_AND_READ;

367.  If not (CAM_COMPLETE) goto 367

368.  If not (CAM_MATCH) goto 486

*****************************************************************
**
**  Lines 369 - 372 performs a read of the event from the
**  cam and a read of the memory pointer from the adjacent
**  RAM.  The event is stored in register 29 and the memory
**  pointer is stored in register 30.
**
*****************************************************************
```

```
369.  CAM_READ;

370.  Reg29 := MBR;

      -  store EVENT in register 29

371.  ADJ_RAM_READ;

      -  read adjacent RAM

372.  Reg30 := MBR;

      -  store MEM_PTR in register 30

*****************************************************************
**
**  Lines 373 - 387 performs an update of the simulation
**  time for the specified LP.  The delay for the LP is
**  stored in register 25 to be used to determine the time
**  tag for the output event.
**
*****************************************************************

373.  Reg21 := BAND(Reg21, Reg52);

374.  Reg21 := BOR(Reg21, Reg29);

375.  Reg31 := BAND(Reg31, Reg52);

376.  Reg31 := BOR(Reg31, R2_MUX(ACC, 0));

      -  store base pointer in register 31

377.  MAR := Reg31;

378.  RAM_READ;

379.  Reg23 := MBR;

      -  store LP_DELAY into register 23

380.  Reg31 := Reg31 + Reg53;

      -  advance pointer
```

```
381.  Reg25 := BAND(Reg25, Reg52);

382.  Reg25 := BOR(Reg25, Reg60);

383.  Reg25 := BOR(Reg25, Reg55);

384.  Reg25 := RSHIFT(Reg25);

      -  register 25 now contains the TIME_TAG

385.  Reg25 := BAND(Reg25, Reg29);

386.  MBR := Reg31;  MAR := Reg25;

387.  RAM_WRITE;

*****************************************************************
**
**  Lines 388 - 405 composes a series of commands that
**  partially formats the output event, obtains the number
**  of arcs for status updating, and searches for another
**  event on the same arc.  If another event is in the CAM,
**  then the status register does not need to be changed.
**
*********************************************************

388.  Reg25 := Reg25 + Reg23;

      -  register 25 now contains the message time including delay

389.  Reg31 := Reg31 + Reg53;

390.  MAR := Reg31;

391.  RAM_READ;

392.  Reg28 := MDR;

393.  Reg31 := Reg31 + Reg53;

394.  Reg23 := BAND(Reg23, Reg52);

395.  Reg23 := BOR(Reg23, Reg28);

396.  Reg28 := BAND(Reg28, Reg60);
```

```
397.  Reg21 := RSHIFT8(Reg21);

398.  Reg21 := BAND(Reg21, Reg55);

399.  Reg23 := RSHIFT8(Reg23);

400.  Reg23 := BAND(Reg23, Reg60);

401.  Reg23 := RSHIFT8(Reg23);

402.  MBR := Reg29;

403.  CAM_SEARCH_TOLP_FROM;

404.  If not (CAM_COMPLETE) goto 404

405.  If CAM_MATCH goto 420

*****************************************************************
**
**  Lines 406 - 420 composes a loop which updates the status
**  register for the specified LP and then checks to see if
**  the memory pointer is a null message.  All 0's signifies
**  a null message.  If the message is null, then a null is
**  sent to all output arcs  and another event is retrieved
**  if it is ready.
**
*****************************************************************

406.  Reg20 := BAND(Reg20, Reg52);

407.  Reg20 := BOR(Reg20, Reg53);

408.  MAR := Reg31;

409.  RAM_READ;

410.  Reg26 := MBR;

411.  ALU := BXOR(Reg26, Reg21);

412.  If ZERO goto 419

413.  Reg31 := Reg31 + Reg53;
```

```
414.  Reg20 := LSHIFT(Reg20);

415.  Reg28 := Reg28 + Reg55;

416.  If ZERO goto 486

417.  JUMP to 408;

419.  R1_MUX(ACC, 1) := BYOR(R1_MUX(ACC, 1), Reg20);

420.  ALU := Reg30;

421.  If ZERO goto 450

*****************************************************************
**
**  Lines 422 - 448 composes a series of instructions which
**  transmits the event, time tag, and memory pointer to the
**  hosts node for processing.  This code is only executed
**  when the event is not a null message.  Register 20
**  contains the event, register 22 contains the interrupt
**  vector, register 25 contains the time tag, and register
**  30 contains the memory pointer.
**
*****************************************************************

422.  Reg20 := BAND(Reg20, Reg52);

423.  Reg20 := BOR(Reg20, Reg55);

424.  Reg20 := BAND(Reg20, Reg21);

425.  Reg28 := BAND(Reg28, Reg52);

426.  Reg28 := BOR(Reg28, Reg27);

427.  Reg28 := BAND(Reg28, Reg59);

428.  Reg20 := BOR(Reg20, Reg28);

429.  Reg20 := Reg20 + Reg53;

430.  If not (WRITE_LOCAL) goto 430
```

```
431.  Reg20 := Reg20 + Reg53;

432.  MBR := Reg20;

433.  Output_Data;

434.  Reg22 := BAND(Reg22, Reg52);

435.  Reg22 := BOR(Reg22, Reg60);

436.  Reg22 := LSHIFT(Reg22);

437.  MBR := Reg22;

438.  Signal_Interrupt;

439.  If not (WRITE_LOCAL) goto 439

440.  MBR := Reg25;

441.  Output_Data;

442.  Write_Local_Toggle;

443.  If not (WRITE_LOCAL_ goto 443

444.  MBR := Reg30;

445.  Output_Data;

446.  Write_Local_Toggle;

447.  In not (WRITE_LOCAL) goto 447

448.  JUMP to 54;

*****************************************************************
**
**  Lines 450 - 486 composes a loop which sends a null
**  message to every output arc because a null message was
**  retrieved.  Register 20 contains the formatted event,
**  register 60 contains the interrupt vector (11111111),
**  and register 25 contains the time tag.
**
*****************************************************************
```

```
450.  Reg31 := BAND(Reg31, Reg52);
451.  Reg31 := BOR(Reg31, R2_MUX(ACC, 0));
452.  Reg31 := Reg31 + Reg53;
453.  Reg23 := BAND(Reg23, Reg52);
454.  Reg31 := Reg31 + Reg53;
455.  MAR := Reg31;
456.  RAM_READ;
457.  Reg28 := MBR;
458.  Reg23 := BOR(Reg23, Reg28);
459.  Reg28 := BAND(Reg28, Reg60);
460.  Reg23 := RSHIFT8(Reg23);
461.  Reg31 := Reg31 + Reg28;
462.  Reg31 := Reg31 + Reg53;
463.  Reg23 := RSHIFT8(Reg23);
464.  Reg23 := BAND(Reg23, Reg60);
465.  MAR := Reg31;
466.  RAM_READ;
467.  Reg20 := MBR;
468.  Reg20 := LSHIFT8(Reg20);
469.  Reg30 := BAND(Reg30, Reg52);
470.  Reg30 := BOR(Reg30, Reg27);
471.  Reg30 := RSHIFT8(Reg30);
```

```
472.  Reg20 := BOR(Reg20, Reg30);

473.  Reg20 := Reg20 + Reg53;

474.  If not (WRITE_LOCAL) goto 474

475.  MBR := Reg20;

476.  Output_Data;

477.  MBR := Reg60;

478.  SIGNAL_INTERRUPT;

479.  If not (WRITE_LOCAL) then goto 479

480.  MBR := Reg25;

461.  Output_Data;

482.  Write_Local_Toggle;

483.  Reg23 := Reg23 + Reg54;

484.  If ZERO goto 54

485.  Reg31 := Reg31 + Reg53;

486.  JUMP to 465;

*****************************************************************
**
**  Lines 487 - 495 are an error routine which signifies
**  that the DES thought an event was ready, but could not
**  retrieve one from the CAM.
**
*****************************************************************

487.  Reg20 := BAND(Reg20, Reg52);

488.  Reg20 := BOR(Reg20, Reg60);

489.  Reg20 := RSHIFT8(Reg20);

490.  If not (WRITE_LOCAL) goto 490
```

```
491.  MBR := Reg20;

492.  Output_Data;

493.  SIGNAL_ERROR;

494.  If not (WRITE_LOCAL) goto 494

495.  JUMP to 54;
```

B.6 Post Event Microcode

```
****************************************************************
**
**  Line 299 shifts the source NODE/LP information over
**  into bits 17 down to 10 as part of message formatting.
**
****************************************************************

299.  Reg27 := RSHIFT8(Reg27);

****************************************************************
**
**  Lines 300 - 308 are used to wait for data, read the data
**  which contains the time tag, advance the RAM pointer,
**  and reset the read_local bit of the status register.
**
****************************************************************

300.  If not (Read_Local) goto 300

301.  If OPCODE goto 260

302.  Reg31 := BAND(Reg31, Reg52);

303.  Reg31 := BOR(Reg31, R2_MUX(ACC, 0));

304.  Reg31 := Reg31 + Reg53;

305.  Input_Data;
```

```
     - this command will enable the data onto the local data bus

     - read it into the DES
     - reset the READ_LOCAL/WRITE_REMOTE bit of the status
register

306.  Reg26 := MBR;

     - Load TIME_TAG into register 26

307.  Read_Local_Toggle;

308.  Reg31 := Reg31 + Reg53;

*****************************************************************
**
**  Lines 309 - 318 are used to read the number of input
**  and output arcs into register 21, store the arc info
** into register 31, mask off the number of output arcs
**  in register 21, advance the RAM pointer to the first
**  output arc, and right shift register 30 so it only
**  contains the number of output arcs.
**
*****************************************************************

309.  MAR := Reg31;

310.  RAM_READ;

311.  Reg21 := MBR;

312.  Reg30 := BAND(Reg30, Reg52);

313.  Reg30 := BOR(Reg30, Reg21);

314.  Reg21 := BAND(Reg21, Reg60);

     - register 21 now contains the #_ARCS_IN

315.  Reg31 := Reg31 + Reg21;

     - advance RAM ptr to start of Output Arcs

316.  Reg31 := Reg31 + Reg53;
```

```
317.  Reg30 := RSHIFT8(Reg30);

318.  Reg30 := RSHIFT8(Reg30);

*************************************************************
**
**  Lines 319 - 342 compose a loop which is used to retrieve
**  output arcs, format the message, and transmit the data
**  to the host processor.  The arc receiving the real
**  message will not be sent a null message.  Line 325
**  checks to ensure the arc receiving the real message is
**  not sent a null message.
**
*************************************************************

319.  Reg28 := BAND(Reg28, Reg52);

320.  Reg28 := BOR(Reg28, Reg60);

321.  Reg28 := RSHIFT8(Reg28);

322.  MAR := Reg31;

323.  RAM_READ;

324.  Reg24 := MBR;

325.  ALU   := BXOR(Reg24, Reg22);

326.  If ZERO then goto 339

327.  Reg24 := LSHIFT8(Reg24);

328.  Reg57 := BXOR(Reg57, Reg59);

329.  Reg57 := BOR(Reg57, Reg24);

330.  If not (WRITE_LOCAL/READ_REMOTE) then goto 330

331.  MBR   := Reg57;

332.   Output_Data;
```

```
333.  MBR    := Reg28;

      -  interrupt vector = 00000011

334.  SIGNAL_INTERRUPT;

335.  If not (WRITE_LOCAL/READ_REMOTE) then goto 335

336.  MBR    := Reg26;

      -  send TIME_TAG out
      -  an interrupt will not be used
      -  the CUBE is expecting an operand

337.  Output_Data;

338.  Write_Local_Toggle;

339.  Reg31 := Reg31 + Reg53;

      -  advance RAM pointer to the next output arc

340.  Reg30 := Reg30 + Reg54;

      -  decrement the number of output arcs left

341.  If ZERO then goto 54

342.  JUMP to 322;
```

Appendix C. *DES Microcode Instruction Set*

Micro Program Instructions

```
 1. R1  := BAND(R1, R2);
 2. R1  := BXOR(R1, R2);
 3. R1  := BOR(R1, R2);
 4. R1  := R1 + R2;
 5. R1  := R1;
 6. ALU := BAND(R1, R2);
 7. ALU := BXOR(R1, R2);
 8. ALU := BOR(R1, R2);
 9. ALU := R1 + R2;
10. ALU := R1;
11. R1 := BAND(R1, R2_MUX(ACC, 0));
12. R1 := BXOR(R1, R2_MUX(ACC, 0));
13. R1 := BOR(R1, R2_MUX(ACC, 0));
14. R1 := R1 + R2_MUX(ACC, 0));
15. R1 := BAND(R1, R2_MUX(ACC, 1));
16. R1 := BXOR(R1, R2_MUX(ACC, 1));
17. R1 := BOR(R1, R2_MUX(ACC, 1));
18. R1 := R1 + R2_MUX(ACC, 1));
19. R1_MUX(ACC,1) := BAND(R1_MUX(ACC, 1), R2);
20. R1_MUX(ACC,1) := BXOR(R1_MUX(ACC, 1), R2);
21. R1_MUX(ACC,1) := BOR(R1_MUX(ACC, 1), R2);
22. R1_MUX(ACC,1) := R1_MUX(ACC, 1) + R2);
23. R1_MUX(ACC,1) := R1_MUX(ACC, 1);

24. R1  := LSHIFT(BAND(R1, R2));
25. R1  := LSHIFT(BXOR(R1, R2));
26. R1  := LSHIFT(BOR(R1, R2));
27. R1  := LSHIFT(R1 + R2);
28. R1  := LSHIFT(R1);
29. R1  := RSHIFT(BAND(R1, R2));
30. R1  := RSHIFT(BXOR(R1, R2));
31. R1  := RSHIFT(BOR(R1, R2));
32. R1  := RSHIFT(R1);
33. R1  := LSHIFT8(BAND(R1, R2));
34. R1  := LSHIFT8(BXOR(R1, R2));
35. R1  := LSHIFT8(BOR(R1, R2));
36. R1  := LSHIFT8(R1 + R2);
37. R1  := LSHIFT8(R1);
```

```
38. R1  := RSHIFT8(BAND(R1, R2));
39. R1  := RSHIFT8(BXOR(R1, R2));
40. R1  := RSHIFT8(BOR(R1, R2));
41. R1  := RSHIFT8(R1 + R2);
42. R1  := RSHIFT8(R1);

43. R1 := LSHIFT(BAND(R1, R2_MUX(ACC, 0)));
44. R1 := LSHIFT(BXOR(R1, R2_MUX(ACC, 0)));
45. R1 := LSHIFT(BOR(R1, R2_MUX(ACC, 0)));
46. R1 := LSHIFT(R1 + R2_MUX(ACC, 0)));
47. R1 := RSHIFT(BAND(R1, R2_MUX(ACC, 0)));
48. R1 := RSHIFT(BXOR(R1, R2_MUX(ACC, 0)));
49. R1 := RSHIFT(BOR(R1, R2_MUX(ACC, 0)));
50. R1 := RSHIFT(R1 + R2_MUX(ACC, 0)));
51. R1 := LSHIFT8(BAND(R1, R2_MUX(ACC, 0)));
52. R1 := LSHIFT8(BXOR(R1, R2_MUX(ACC, 0)));
53. R1 := LSHIFT8(BOR(R1, R2_MUX(ACC, 0)));
54. R1 := LSHIFT8(R1 + R2_MUX(ACC, 0)));
55. R1 := RSHIFT8(BAND(R1, R2_MUX(ACC, 0)));
56. R1 := RSHIFT8(BXOR(R1, R2_MUX(ACC, 0)));
57. R1 := RSHIFT8(BOR(R1, R2_MUX(ACC, 0)));
58. R1 := RSHIFT8(R1 + R2_MUX(ACC, 0)));

59. R1 := LSHIFT(BAND(R1, R2_MUX(ACC, 1)));
60. R1 := LSHIFT(BXOR(R1, R2_MUX(ACC, 1)));
61. R1 := LSHIFT(BOR(R1, R2_MUX(ACC, 1)));
62. R1 := LSHIFT(R1 + R2_MUX(ACC, 1)));
63. R1 := RSHIFT(BAND(R1, R2_MUX(ACC, 1)));
64. R1 := RSHIFT(BXOR(R1, R2_MUX(ACC, 1)));
65. R1 := RSHIFT(BOR(R1, R2_MUX(ACC, 1)));
66. R1 := RSHIFT(R1 + R2_MUX(ACC, 1)));
67. R1 := LSHIFT8(BAND(R1, R2_MUX(ACC, 1)));
68. R1 := LSHIFT8(BXOR(R1, R2_MUX(ACC, 1)));
69. R1 := LSHIFT8(BOR(R1, R2_MUX(ACC, 1)));
70. R1 := LSHIFT8(R1 + R2_MUX(ACC, 1)));
71. R1 := RSHIFT8(BAND(R1, R2_MUX(ACC, 1)));
72. R1 := RSHIFT8(BXOR(R1, R2_MUX(ACC, 1)));
73. R1 := RSHIFT8(BOR(R1, R2_MUX(ACC, 1)));
74. R1 := RSHIFT8(R1 + R2_MUX(ACC, 1)));

75. R1_MUX(ACC,1) := LSHIFT(BAND(R1_MUX(ACC, 1), R2));
76. R1_MUX(ACC,1) := LSHIFT(BXOR(R1_MUX(ACC, 1), R2));
77. R1_MUX(ACC,1) := LSHIFT(BOR(R1_MUX(ACC, 1), R2));
78. R1_MUX(ACC,1) := LSHIFT(R1_MUX(ACC, 1) + R2));
79. R1_MUX(ACC,1) := LSHIFT(R1_MUX(ACC, 1));
```

```
 80. R1_MUX(ACC,1) := RSHIFT(BAND(R1_MUX(ACC, 1), R2));
 81. R1_MUX(ACC,1) := RSHIFT(BXOR(R1_MUX(ACC, 1), R2));
 82. R1_MUX(ACC,1) := RSHIFT(BOR(R1_MUX(ACC, 1), R2));
 83. R1_MUX(ACC,1) := RSHIFT(R1_MUX(ACC, 1) + R2));
 84. R1_MUX(ACC,1) := RSHIFT(R1_MUX(ACC, 1));
 85. R1_MUX(ACC,1) := LSHIFT8(BAND(R1_MUX(ACC, 1), R2));
 86. R1_MUX(ACC,1) := LSHIFT8(BXOR(R1_MUX(ACC, 1), R2));
 87. R1_MUX(ACC,1) := LSHIFT8(BOR(R1_MUX(ACC, 1), R2));
 88. R1_MUX(ACC,1) := LSHIFT8(R1_MUX(ACC, 1) + R2));
 89. R1_MUX(ACC,1) := LSHIFT8(R1_MUX(ACC, 1));
 90. R1_MUX(ACC,1) := RSHIFT8(BAND(R1_MUX(ACC, 1), R2));
 91. R1_MUX(ACC,1) := RSHIFT8(BXOR(R1_MUX(ACC, 1), R2));
 92. R1_MUX(ACC,1) := RSHIFT8(BOR(R1_MUX(ACC, 1), R2));
 93. R1_MUX(ACC,1) := RSHIFT8(R1_MUX(ACC, 1) + R2));
 94. R1_MUX(ACC,1) := RSHIFT8(R1_MUX(ACC, 1));

 95. MAR := R2; MBR := R1;
 96. R1  := MBR;
 97. MBR := R1;
 98. SIGNAL_INTR(DATA);
120.  MAR := R2;

STATUS COMMANDS

 99. SIGNAL_READY;
100. SIGNAL_ERROR;
101. READ_LOCAL/WRITE_REMOTE;
102. WRITE_LOCAL/READ_REMOTE;

MSL CHECKS

103. IF OPCODE THEN GOTO R1/R2
104. IF NOT (OPCODE and READ_LOCAL/WRITE_REMOTE) THEN GOTO R1/R2
105. IF ZERO THEN GOTO R1/R2
106. IF NOT (READ_LOCAL/WRITE_REMOTE) THEN GOTO R1/R2
107. IF NOT (WRITE_LOCAL/READ_REMOTE) THEN GOT R1/R2
108. IF CAM_MATCH THEN GOT R1/R2
109. IF MIN_COMPLETE THEN GOTO R1/R2
110. JUMP TO R1/R2
111. JUMP TO IR(MAPPING_ROM)
131. IF NOT (CAM_MATCH) THEN GOTO R1/R2
132. IF NOT (CAM_COMPLETE) THEN GOTO R1/R2

RAM INSTRUCTIONS
```

```
112. RAM_WRITE(1);
113. RAM_WRITE(2);
114. RAM_WRITE(3);
115. RAM_WRITE(4);
116. RAM_READ(1);
117. RAM_READ(2);
118. RAM_READ(3);
119. RAM_READ(4);
```

Content-Addressable Memory Instructions

```
121. CAM_INIT;
122. CAM_MIN_FIND_AND_READ;
123. CAM_SEARCH_TOLP_FROM
124. CAM_WRITE_WORD
125. CAM_RESERVE_ARC
126. CAM_READ;
127. ADJ_RAM_WRITE;
128. ADJ_RAM_READ;
```

DATA TRANSFER WITH CUBE

```
129. INPUT_DATA;
130. OUTPUT_DATA;
```

Appendix D. *DES VHDL Behavioral and Structural Code*

This appendix contains the a complete behavioral VHDL listing of all the files used in the DES coprocessor. All of the VHDL files were written using Synopsys VHDL. A partial structural listing is also included, but all of the components in the DES are not at the structural level. The source code is listed in volume 2 of this research effort. A copy of volume 2 can be requested through the VLSI Lab, Department of Electrical and Computer Engineering within the School of Engineering.

References

1. Banton, David W., PhD Candidate, "Personal Conversation," July-August 1992.
2. Brothers, Charles P., PhD Candidate, "Personal Conversation," July-August 1992.
3. Catlin, Gary and Bill Paseman. "Hardware Acceleration of Logic Simulation Using a Data Flow Architecture." *International Conference on Computer-Aided Design*. 130–132. Washington D.C.: IEEE, 1985.
4. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*:198–206 (April 1981).
5. d'Abreu, Manuel A. "Gate-Level Simulation," *IEEE Design and Test*, *2*:63–71 (December 1985).
6. Franklin, M. A. and other "Parallel Machines and Algorithms for Discrete-Event Simulation." *International Conference on Parallel Processing*. 449–458. Columbus, Oh.: IEEE, 1984.
7. Fujimoto, Richard M. and others. "The Roll Back Chip: Hardware Support for Distributed Simulation Using Time Warp," *Distributed Simulation*, *19*:81–86 (February 1988).
8. Georing, Richard. "Simulation accelerators address throughput issues," *Computer Design*, 42–47 (March 1988).
9. Intel Corporation, Mt. Prospect, IL. *Microprocessors, Volume II*, 1991.
10. Jefferson, David. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7:404–425 (July 1985).
11. Kesting, Loren F. *Final Report: A User's Manual for OCTTOOLS*. The Air Force Institute of Technology (AU), Wright-Patterson AFB, OH. EENG699.
12. Lee, Ann Kathryn. *An Empirical Study of Combining Communicating Processes in a Parallel Discrete Event Simulation*. MS thesis, AFIT/GCS/ENG/90D-08, Air Force Institue of Technology (AU), Wright-Patterson AFB, OH, December 1990.
13. Misra, Jayadev. "Distributed Discrete-Event Simulation," *ACM Computering Surveys*, *18*:39–65 (March 1986).
14. Neelamkavil, Francis. *Computer Simulation and Modelling*. John Wiley and Sons, 1987.
15. Nicol, David M. and Jr. Paul F. Reynolds. "An Efficient Framework for Parallel Simulations." *SCS Multiconference, PADS Workshop*. 167–173. 1991.
16. Pritsker, A. Alan B. and Claude D. Pegden. *Introduction to Simulation and SLAM*. John Wiley and Sons, 1984.
17. Reed, Daniel A. and Allen D. Malony. "Parallel Discrete Event Simulation: The Chandy-Misra Approach." *Distributed Simulation*. 8–13. La Jolla CA: SCS, 1988.
18. Synopsys, Inc. *Design Compiler Reference Manual, Version 2.2*, October 1991.

19. Synopsys, Inc. *Simulation Graphical Environment User's Guide, Version 2.2*, October 1991.

20. Tanenbaum, Andrew S. *Structured Computer Organization, 3rd Edition*. Prentice Hall, 1990.

21. Taylor, Paul J. *Requirements Analysis for a Hardware, Discrete-Event, Simulation Engine Accelerator*. MS thesis, AFIT/GCE/ENG/91D-11, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1991.

22. Van Horn, Prescott J. *Development of a Protocol User's Guideline for Conservative Parallel Simulations*. MS thesis, AFIT/GCS/ENG/92D-19, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

23. Wieland, Frederick and others. "Speedup Bias." Unpublished Paper.

Vita

Captain David W. Daniel was born on 24 July 1963 at Barksdale AFB, Louisiana. He graduated from Warrensburg High School in 1981. He then received his undergraduate computer science degree from Central Missouri State University in 1985. He received his Air Force commission on 2 October 1986 and served four and a half years in the Communications-Computer Systems Directorate at the Air Force Institute of Technology (AFIT). He then entered the AFIT in-residence program to receive his masters in computer engineering.

Permanent address: 2805 Arden Ave
Dayton, Ohio 45420

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
DESIGN OF A HARDWARE DISCRETE EVENT SIMULATION COPROCESSOR

5. FUNDING NUMBERS

6. AUTHOR(S)
David W. Daniel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/93M-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA (LTC John Toole)
3701 N. Fairfax Dr.
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A hardware discrete event simulation (DES) coprocessor was designed to eliminate synchronization overhead as a possible bottleneck. The target architecture is an eight node Intel iPSC/2 Hypercube, but this design has application to future CPU designs that wish to incorporate on-chip architectural features to better support parallel processor synchronization. A structural description of a general-purpose DES hardware coprocessor is given with approximately 90 percent of the components written at the gate level. The remaining components use low-level behavioral descriptions. While the DES coprocessor microcode implements the Chandy-Misra protocol, general-purpose support for a wide-range of protocols was a primary hardware design objective.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Simulation, Parallel Processing, Discrete Event Simulation, VHDL, Coprocessor, Simulation Accelerator	149
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL